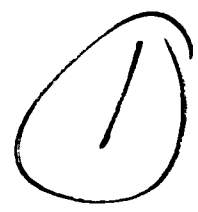


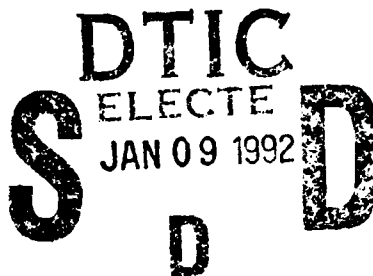
AD-A244 978



**SOFTWARE DESIGN DOCUMENT
GT Real-Time Software Host CSCI (9B)**

Volume 1 of 2 Sections 1.0 - 2.12.19.2

June, 1991



Prepared by:

BBN Systems and Technologies,
A Division of Bolt Beranek and Newman Inc.
10 Moulton Street
Cambridge, MA 02138
(617) 873-3000 FAX: (617) 873-4315

Prepared for:

Defense Advanced Research Projects Agency (DARPA)
Information and Science Technology Office
1400 Wilson Blvd., Arlington, VA 22209-2308
(202) 694-8232, AUTOVON 224-8232

Program Manager for Training Devices (PM TRADE)
12350 Research Parkway
Orlando, FL 32826-3276
(407) 380-4518

92-00253



**APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED**

92 1 6 061

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE June 1991	3. REPORT TYPE AND DATES COVERED Software Design Document
4. TITLE AND SUBTITLE Software Design Document GT Real-Time Software Host CSCI (9B)			5. FUNDING NUMBERS Contract Numbers: MDA972-89-C-0060 MDA972-89-C-0061
6. AUTHOR(S) Author not specified.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Bolt Beranek and Newman, Inc. (BBN) Systems and Technologies; Advanced Simulation 10 Moulton Street Cambridge, MA 02138			8. PERFORMING ORGANIZATION REPORT NUMBER Advanced Simulation #: 9117
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency (DARPA) 3701 North Fairfax Drive Arlington, VA 22203-1714			10. SPONSORING/MONITORING AGENCY REPORT NUMBER DARPA Report Number: None.
11. SUPPLEMENTARY NOTES None			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution Statement A: Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE Distribution Code: A
13. ABSTRACT (Maximum 200 words) A Simulation Network (SIMNET) project Software Design Document that describes the GT Real-Time Software Host Computer Software Configuration Item (CSCI number 9B) of the SIMNET hardware and software training system for vehicle crew training and operational training.			
14. SUBJECT TERMS SIMNET Software Design Document for the GT Real-Time Software Host CSCI (CSCI 9B).			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Same as report.

2

**SOFTWARE DESIGN DOCUMENT
GT Real-Time Software Host CSCI (9B)**

Volume 1 of 2 Sections 1.0 - 2.12.19.2

June, 1991

Prepared by:

BBN Systems and Technologies,
A Division of Bolt Beranek and Newman Inc.
10 Moulton Street
Cambridge, MA 02138
(617) 873-3000 FAX: (617) 873-4315

Prepared for:

Defense Advanced Research Projects Agency (DARPA)
Information and Science Technology Office
1400 Wilson Blvd., Arlington, VA 22209-2308
(202) 694-8232, AUTOVON 224-8232

Program Manager for Training Devices (PM TRADE)
12350 Research Parkway
Orlando, FL 32826-3276
(407) 380-4518



Accession For	
NTIS CRAS!	✓
DTIC TAB	
Unannounced Justification	
By	
Date	
A-11	

**APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED**

Table of Contents

1	INTRODUCTION: GT REAL-TIME SOFTWARE HOST CSCI	1
1.1	THE SIMULATOR	1
1.1.1	The Simulation Host	2
1.1.2	The CIG	2
1.2	CIG-SIM COMMUNICATION	2
1.3	RTSW SOFTWARE STRUCTURE	3
1.5	HOW THIS DOCUMENT IS ORGANIZED	5
2	CSC DESCRIPTIONS	7
2.1	TASK INITIALIZATION (/CIG/GTBINSRC)	9
2.1.1	bx147_main.c	9
2.1.1.1	main	9
2.1.1.2	poll_shutdown	10
2.1.2	mkcal.c	11
2.1.2.1	make_cal_overlay	11
2.1.2.2	make_cal_matrices	12
2.1.2.3	make_cal_patterns	13
2.1.2.4	pix_mult	13
2.1.3	rtt.c	14
2.1.3.1	main	14
2.1.3.2	check_restart	17
2.1.3.3	disable_restart	17
2.1.3.4	rtt_shutdown	18
2.1.3.5	poll_shutdown	19
2.2	2-D OVERLAY COMPILER (/CIG/LIBSRC/LIB2D)	20
2.2.1	bit_blt.c (setup_bit_blt)	26
2.2.2	cig_2d_setup.c	26
2.2.3	cig_comp_2d.c (compile_2d)	27
2.2.4	cig_getm_2d.c (get_msg_2d)	28
2.2.5	cig_link_2d.c (linkup)	29
2.2.6	comp.c (setup_comp_start)	30
2.2.7	draw_line.c (setup_draw_line)	31
2.2.8	get_thing.c	32
2.2.9	init_stuff.c	33
2.2.10	oval_rect.c (setup_oval_rectangle)	33
2.2.11	poly.c (setup_poly)	34

2.2.12	proc_cmd.c (process_command).....	35
2.2.13	string.c (setup_define_string)	36
2.2.14	text.c (setup_text)	37
2.2.15	u_comp_2d.c (compile_2d)	38
2.2.16	u_getm_2d.c (get_msg_2d)	38
2.2.17	u_link_2d.c (linkup)	39
2.2.18	u_main2d.c (main)	39
2.2.19	window.c (setup_define_window)	40
2.3	BACKEND MANAGER (/CIG/LIBSRC/LIBBACKEND)	42
2.3.1	aa_init.c	43
2.3.1.1	active_area_init	43
2.3.1.2	clear	44
2.3.1.3	extended_ram_available	44
2.3.2	backend_branch.c (backend_set_branch)	45
2.3.3	backend_color.c (backend_set_color)	45
2.3.4	backend_laser.c	46
2.3.4.1	backend_laser_request_range.....	47
2.3.4.2	backend_response.....	48
2.3.5	backend_man.c.....	49
2.3.5.1	backend_setup.....	49
2.3.5.2	backend_reset.....	50
2.3.5.3	backend_sim_init.....	51
2.3.5.4	backend_send_req.....	51
2.3.5.5	backend_get_object_addr.....	52
2.3.5.6	backend_clear_laser_requests.....	52
2.3.6	backend_paths.c.....	53
2.3.6.1	backend_set_paths	53
2.3.6.2	backend_update_view_paths.....	53
2.3.7	backend_thermal.c (backend_set_thermal).....	54
2.3.8	backend_video.c (backend_set_video)	55
2.3.9	dl_man.c (dl_setup).....	55
2.3.10	ppm_obj.c.....	56
2.3.10.1	ppm_setup.....	56
2.3.10.2	ppm_init	57
2.3.10.3	ppm_get_data	58
2.3.10.4	ppm_load.....	58
2.3.10.5	gos_ppm_query	59
2.3.10.6	gos_ppm_query_menu	60

2.4	BALLISTICS PROCESSING (/CIG/LIBSRC/LIBBALL).....	62
2.4.1	Ballistics Mainline.....	68
2.4.1.1	bx_init.c.....	68
2.4.1.2	bx_probe.c	69
2.4.1.3	bx_reset.c	71
2.4.1.4	bx_task.c.....	72
	2.4.1.4.1 bx_task	72
	2.4.1.4.2 bx_task_cleanup.....	73
2.4.2	Ballistics Interface Message Processing.....	74
2.4.2.1	b0_aam_centroid.c.....	75
2.4.2.2	b0_aam_sw_corner.c.....	75
2.4.2.3	b0_add_static_vehicle.c	75
2.4.2.4	b0_add_traj_table.c	76
2.4.2.5	b0_bal_config.c	77
2.4.2.6	b0_bvol_entry.c.....	77
2.4.2.7	b0_cancel_round.c.....	78
2.4.2.8	b0_cig_frame_rate.c	79
2.4.2.9	b0_database_info.c.....	79
2.4.2.10	b0_delete_static_vehicle.c.....	80
2.4.2.11	b0_delete_traj_table.c.....	81
2.4.2.12	b0_error_detected.c.....	81
2.4.2.13	b0_inapp_message.c.....	81
2.4.2.14	b0_lm_read.c	81
2.4.2.15	b0_model_directory.c.....	82
2.4.2.16	b0_model_entry.c	82
2.4.2.17	b0_new_frame.c.....	83
2.4.2.18	b0_print.c	83
2.4.2.19	b0_process_chord.c.....	84
2.4.2.20	b0_process_round.c	85
2.4.2.21	b0_round_fired.c.....	86
2.4.2.22	b0_state_control.c.....	87
2.4.2.23	b0_status_request.c.....	88
2.4.2.24	b0_tf_init_hdr.c	88
2.4.2.25	b0_tf_init_pt.c	89
2.4.2.26	b0_tf_state.c	89
2.4.2.27	b0_tf_vehicle_pos.c.....	90

2.4.2.28	b0_traj_chord.c.....	91
2.4.2.29	b0_traj_entry.c.....	92
2.4.2.30	b0_undefined_message.c.....	93
2.4.3	Ballistics Database Interaction	94
2.4.3.1	bx_bvol_int.c.....	94
2.4.3.2	bx_chord_intersect.c.....	95
2.4.3.3	bx_compute_round.c	96
2.4.3.3.1	bx_return_miss	97
2.4.3.3.2	bx_guntip_within_db.....	97
2.4.3.3.3	bx_find_shot_report	98
2.4.3.3.4	bx_round_tracer_position	99
2.4.3.3.5	bx_find_round_hit	100
2.4.3.4	bx_find_vehicle.c	100
2.4.3.5	bx_functions.c.....	101
2.4.3.5.1	bx_new_round.....	102
2.4.3.5.2	bx_delete_round.....	102
2.4.3.5.3	bx_get_db_pos	103
2.4.3.5.4	bx_get_chord_end.....	103
2.4.3.5.5	bx_new_bvol	104
2.4.3.5.6	bx_free_lm_cache	104
2.4.3.5.7	bx_new_poly	105
2.4.3.5.8	bx_get_lb_from_lm.....	105
2.4.3.5.9	bx_new_stat_veh.....	105
2.4.3.5.10	bx_delete_stat_veh.....	106
2.4.3.5.11	bx_dist_sq_pt_line.....	106
2.4.3.6	bx_get_lm_data.c.....	107
2.4.3.7	bx_get_lm_grid.c.....	108
2.4.3.8	bx_model_int.c	108
2.4.3.9	bx_poly_int.c.....	109
2.4.3.10	bx_tf_pack.c	110
2.4.3.10.1	bx_tf_init_pt_cache	111
2.4.3.10.2	bx_tf_pts	111
2.4.3.10.3	bx_tf_next.....	111
2.4.3.10.4	bx_tf_copy_msg.....	112
2.4.3.10.5	bx_tf_pt_data	112
2.4.3.10.6	bx_tf_new_tf_pts.....	113

	2.4.3.10.7	bx_tf_free_tf_pts	113
	2.4.3.11	bx_trajectory.c	114
	2.4.3.12	shot_report.c	114
2.4.4	Ballistics Message Queue Management.....		116
	2.4.4.1	mx_error.c	116
	2.4.4.2	mx_open.c	117
	2.4.4.3	mx_peek.c	117
	2.4.4.4	mx_push.c	118
	2.4.4.5	mx_skip.c	120
	2.4.4.6	mx_wcopy.c	120
2.5	CIG CONFIGURATION (/CIG/LIBSRC/LIBCONFIG)		122
2.5.1	aam_manager.c		122
	2.5.1.1	aam_malloc	123
	2.5.1.2	return_aam_ptr	123
	2.5.1.3	system_aam_init	124
	2.5.1.4	dynamic_aam_init	124
2.5.2	bbnctype.c		125
2.5.3	cig_config.c		125
2.5.4	fill_tree.c		128
	2.5.4.1	fill_tree	128
	2.5.4.2	power	129
2.5.5	get_vehicle_position.c		129
2.5.6	getch.c		129
2.5.7	overlay_setup.c		130
2.5.8	process_vflags.c		131
2.5.9	process_vppos.c		131
2.6	ESIFA INTERFACE (/CIG/LIBSRC/LIBESIFA).....		134
2.6.1	esifa_fade.c (esifa_set_fade).....		135
2.6.2	esifa_laser.c		136
	2.6.2.1	esifa_laser_request_range	136
	2.6.2.2	esifa_laser_return	137
2.6.3	esifa_load.c		138
2.6.4	esifa_man.c		139
	2.6.4.1	esifa_setup	139
	2.6.4.2	esifa_sim_init	140
	2.6.4.3	esifa_send_req	140
	2.6.4.4	esifa_get_object_addr	141
	2.6.4.5	esifa_ConfigData	141

2.6.4.6	esifa_queue_data	142
2.6.4.7	esifa_queue_download	143
2.6.4.8	esifa_send_queue.....	144
2.6.5	esifa_query.c	144
2.6.5.1	esifa_read_ports	144
2.6.5.2	esifa_write_ports	145
2.6.5.3	esifa_read.....	146
2.6.5.4	esifa_write.....	147
2.6.5.5	esifa_download.....	148
2.6.6	esifa_special.c (esifa_set_special)	149
2.6.7	esifa_thermal.c (esifa_set_thermal)	149
2.6.8	esifa_video.c (esifa_set_video)	150
2.7	STAND-ALONE HOST EMULATOR (/CIG/LIBSRC/LIBFLEA)	152
2.7.1	autopilot.c	154
2.7.2	cf_translator.c	155
2.7.2.1	config_translator.....	156
2.7.2.2	process_cig_ctl	157
2.7.2.3	process_file_description.....	158
2.7.2.4	process_configtree_node	159
2.7.2.5	process_viewport_state.....	159
2.7.2.6	process_define_tx_mode	160
2.7.2.7	process_overlay_setup.....	161
2.7.2.8	process_tf_init_hdr.....	162
2.7.2.9	process_tf_init_pt	163
2.7.2.10	process_agl_setup.....	163
2.7.2.11	process_sio_init	164
2.7.2.12	process_add_traj_table.....	165
2.7.2.13	process_traj_entry.....	166
2.7.2.14	process_2d_setup	166
2.7.2.15	process_lt_state.....	167
2.7.2.16	process_dr11_pkt_size	168
2.7.2.17	process_sio_close	169
2.7.2.18	process_tf_state.....	169
2.7.2.19	process_ammo_define	170
2.7.2.20	process_ppm_display_mode.....	171
2.7.2.21	process_ppm_display_offset.....	172
2.7.2.22	process_ppm_pixel_location.....	173

	2.7.2.23	process_ppm_pixel_state.....	173
	2.7.2.24	read_a_keyword.....	174
	2.7.2.25	remove_white_space.....	175
	2.7.2.26	remove_comment_lines.....	176
2.7.3		dynamic_demo.c	176
2.7.4		encode_routines.c.....	177
	2.7.4.1	upd_matrix_values.....	178
	2.7.4.2	upd_rotation_values.....	179
	2.7.4.3	upd_dynamic_matrix.....	180
	2.7.4.4	upd_view_flags.....	180
	2.7.4.5	upd_round_fired.....	181
	2.7.4.6	upd_chord_fired.....	182
	2.7.4.7	upd_auto_fire.....	182
	2.7.4.8	upd_rem_static_veh.....	183
	2.7.4.9	upd_add_static_veh.....	184
	2.7.4.10	upd_send_dynamic.....	184
	2.7.4.11	upd_count_hits_per_min.....	185
	2.7.4.12	upd_view_mode.....	185
	2.7.4.13	upd_show_eff.....	185
	2.7.4.14	upd_clouds.....	186
	2.7.4.15	upd_req_agl.....	186
	2.7.4.16	upd_req_point.....	187
	2.7.4.17	upd_req_lrange.....	188
	2.7.4.18	upd_view_mag.....	188
	2.7.4.19	upd_subsys_mode.....	189
	2.7.4.20	upd_viewport_up.....	189
	2.7.4.21	upd_send_stop.....	190
	2.7.4.22	flea_error_print.....	191
	2.7.4.23	put_in_hdr.....	191
	2.7.4.24	upd_flea_vehicles.....	192
	2.7.4.25	fire_round.....	193
	2.7.4.26	process_chord.....	193
	2.7.4.27	process_round.....	194
	2.7.4.28	cancel_round.....	195
	2.7.4.29	upd_sio_write.....	195
	2.7.4.30	upd_lt_state.....	196

	2.7.4.31	send_gun_overlay.....	196
	2.7.4.32	send_ammo_define.....	197
2.7.5		flea.c.....	197
	2.7.5.1	flea	198
	2.7.5.2	flea_io_task	200
	2.7.5.3	flea_getchar	200
	2.7.5.4	flea_dummy_getchar.....	201
	2.7.5.5	flea_IO_mode.....	202
	2.7.5.6	flea_IO_on	202
	2.7.5.7	flea_IO_off.....	203
	2.7.5.8	flea_initialized.....	203
	2.7.5.9	scratch_flea.....	204
	2.7.5.10	flea_printf	204
	2.7.5.11	flea_cleanup.....	204
	2.7.5.12	flea_io_task_cleanup.....	205
2.7.6		flea_agl_terrain_follow.c	205
2.7.7		flea_agpt_locations.c	206
	2.7.7.1	flea_agpt_locations.....	206
	2.7.7.2	flea_agpt_locations_main_menu.....	208
	2.7.7.3	new_pos_orient.....	209
	2.7.7.4	update_dyn_demo.....	209
2.7.8		flea_agpt_switches.c.....	210
	2.7.8.1	flea_agpt_switches.....	210
	2.7.8.2	set_command_2d.....	213
	2.7.8.3	update_mag	214
	2.7.8.4	update_subsys_mode	214
	2.7.8.5	update_vpt.....	215
	2.7.8.6	flea_agpt_switches_main_menu.....	215
	2.7.8.7	derror	216
2.7.9		flea_atp.c.....	216
	2.7.9.1	flea_atp	217
	2.7.9.2	flea_atp_main_menu	218
2.7.10		flea_bal_opts.c	218
	2.7.10.1	flea_bal_opts.....	218
	2.7.10.2	flea_bal_opts_main_menu.....	222
2.7.11		flea_db_traverse.c	222
2.7.12		flea_decode_data.c	223

2.7.13	flea_demo.c	226
2.7.14	flea_draw_2d.c	226
2.7.15	flea_encode_data.c	227
2.7.16	flea_graphics_test.c.....	228
2.7.16.1	flea_graphics_test	228
2.7.16.2	flea_graphics_test_main_menu	230
2.7.17	flea_init_cig_sw.c	230
2.7.18	flea_ppm_obj.c.....	233
2.7.18.1	upd_ppm	233
2.7.18.2	flea_calibration_image.....	235
2.7.18.3	flea_ppm_display_mode.....	236
2.7.18.4	flea_ppm_display_offset	237
2.7.18.5	flea_ppm_pixel_location.....	237
2.7.18.6	flea_ppm_pixel_state.....	238
2.7.19	flea_script.c.....	238
2.7.19.1	flea_abs_playback.....	239
2.7.19.2	get_next_packet.....	240
2.7.19.3	get_next_message	240
2.7.20	flea_simulate_vehicles.c.....	241
2.7.20.1	flea_simulate_vehicles	241
2.7.20.2	find_pitch_and_roll	243
2.7.21	flea_switches.c	244
2.7.21.1	flea_switches.....	244
2.7.21.2	flea_switches_main_menu	247
2.7.22	flea_update_pos.c	248
2.7.23	flea_vch_control.c	249
2.7.23.1	flea_vch_control.....	249
2.7.23.2	flea_vch_control_main_menu.....	251
2.7.24	get_sio_data.c (get_sio_write_data)	252
2.7.25	model_demo.c	253
2.7.26	tick.c.....	254
2.7.26.1	tick_init	255
2.7.26.2	tick	255
2.7.26.3	flea_prompt	259
2.7.26.4	tick_main_menu	259
2.7.26.5	menu_header.....	260
2.7.26.6	update_menu_header.....	261
2.7.27	tick_ppm.c.....	261

2.7.27.1	tick_ppm.....	262
2.7.27.2	tick_ppm_menu.....	264
2.7.27.3	tick_ppm_menu_header	265
2.7.27.4	tick_ppm_update_info	265
2.7.28	tick_script.c.....	266
2.7.28.1	tick_script	266
2.7.28.2	tick_script_main_menu	267
2.7.29	update_2d.c	268
2.7.30	update_agpt_2d.c	268
2.8	DTP COMMAND GENERATOR (/CIG/LIBSRC/LIBGENDTP)	270
2.8.1	dtp_compiler.c	271
2.8.2	dtp_funcs.c	272
2.8.2.1	push_node.....	272
2.8.2.2	pop_node.....	273
2.8.2.3	what_node_on_stack.....	273
2.8.2.4	init_dtp_stacks.....	274
2.8.2.5	dtp_malloc.....	274
2.8.2.6	dtp_malloc_init	274
2.8.3	dtp_trav1.c	275
2.8.4	dtp_trav2.c	277
2.8.5	rcfuncs.c	278
2.8.5.1	rcl_init_stack.....	279
2.8.5.2	rcl_push	280
2.8.5.3	rcl_pop.....	280
2.8.5.4	rcl_patch_adrs	281
2.8.5.5	rcl_set_errptr.....	281
2.8.5.6	rcl_init_adrs.....	282
2.8.5.7	rcl_rtn_adrs	282
2.8.5.8	rcl_set_label.....	283
2.8.5.9	rcl_set_cntlbl.....	283
2.8.5.10	rcl_lblcmd	284
2.8.5.11	rcl_command.....	285
2.8.5.12	rcl_component.....	287
2.8.5.13	rcl_data	288
2.8.5.14	rcl_stuff_data	289
2.9	USER INTERFACE MODE (/CIG/LIBSRC/LIBGOSSIP).....	291
2.9.1	agpt_statistics.c	292

2.9.2	buffer_errors.c	292
2.9.3	ded_object.c	292
2.9.3.1	ded_setup.....	293
2.9.3.2	ded_init_mdl_addr	294
2.9.3.3	ded_add_model.....	294
2.9.3.4	ded_add_effect.....	295
2.9.3.5	ded_model_end_addr	295
2.9.3.6	ded_adjust_addr_tables.....	296
2.9.3.7	ded_uninit	296
2.9.3.8	ded_sub_end.....	297
2.9.3.9	ded_cross_border	297
2.9.3.10	ded_relocate_model	298
2.9.3.11	ded_process_directory.....	298
2.9.3.12	ded_dtp_trace	300
2.9.3.13	ded_load_dtp_code.....	300
2.9.3.14	ded_model_offset	301
2.9.3.15	ded_print_tables.....	302
2.9.3.16	ded_object_debug	302
2.9.4	dtp_emu.c.....	303
2.9.4.1	dtp_emu	303
2.9.4.2	display	304
2.9.4.3	outdisplay	305
2.9.4.4	hxflt.....	305
2.9.4.5	hexdisplay.....	306
2.9.4.6	fioh.....	306
2.9.4.7	htof.....	307
2.9.4.8	mat_mult.....	307
2.9.4.9	get_lm.....	308
2.9.5	gos_120tx.c	308
2.9.6	gos_bal_query.c	310
2.9.7	gos_db_query.c	310
2.9.7.1	gos_db_query.....	310
2.9.7.2	gos_display_db_info.....	312
2.9.7.3	gos_db_query_menu.....	312
2.9.8	gos_locate.c	313
2.9.9	gos_memory.c	313
2.9.10	gos_model.c	314

2.9.10.1	gos_model.....	315
2.9.10.2	rcl_set_modloc.....	317
2.9.11	gos_mpv.c	318
2.9.12	gos_mpvio.c	320
2.9.13	gos_polys.c	322
2.9.14	gos_system.c	323
2.9.15	gossip.c	325
2.9.15.1	gossip.....	326
2.9.15.2	gossip_tick.....	327
2.9.15.3	gos_timing_printout.....	329
2.9.15.4	s_step.....	330
2.9.15.5	gos_single_step.....	331
2.9.15.6	gos_prompt	331
2.9.15.7	gos_main_menu	332
2.9.15.8	gos_getchar	332
2.9.15.9	gos_dummy_getchar.....	333
2.9.15.10	gos_IO_on	333
2.9.15.11	gos_IO_off.....	333
2.9.15.12	gossip_cleanup	334
2.9.16	make_bbn_logo.c	334
2.9.17	mx3_hword.c.....	335
2.9.17.1	mx3_open	335
2.9.17.2	mx3_push	336
2.9.17.3	mx3_peek	337
2.9.17.4	mx3_skip.....	338
2.9.17.5	mx3_error.....	338
2.9.17.6	mx3_hwcopy	339
2.9.18	replace_mod.c.....	339
2.9.18.1	replace_mod.....	339
2.9.18.2	single_lite.....	340
2.9.18.3	double_lite.....	341
2.9.18.4	triple_lite.....	342
2.9.18.5	vasi_lite.....	342
2.9.18.6	outahere.....	343
2.9.19	test_commands.c	344
2.10	HOST INTERFACE MANAGER (/CIG/LIBSRC/LIBHOST).....	346
2.10.1	host_dr11_if.c.....	347

2.10.1.1	open_dr11_interface	347
2.10.1.2	exchange_dr11_data	348
2.10.1.3	exchange_dr11_data_sim.....	349
2.10.1.4	init_dr11_interface	349
2.10.2	host_enet_if.c.....	350
2.10.2.1	open_enet_interface	350
2.10.2.2	exchange_enet_data	351
2.10.2.3	exchange_enet_data_sim	352
2.10.2.4	slave_cig_enet_sync	353
2.10.2.5	init_enet_interface	354
2.10.3	host_flea_if.c.....	355
2.10.3.1	open_flea_interface.....	355
2.10.3.2	exchange_flea_data.....	356
2.10.3.3	flea_host_if.....	357
2.10.3.4	init_flea_interface.....	358
2.10.4	host_if_debug.c.....	358
2.10.4.1	msg_shell_sort.....	359
2.10.4.2	host_enable_all_debug_msgs.....	360
2.10.4.3	host_disable_all_debug_msgs	360
2.10.4.4	host_if_debug_init	361
2.10.4.5	clear_line	361
2.10.4.6	host_if_display_enabled_msgs	362
2.10.4.7	host_list_msgs.....	362
2.10.4.8	host_if_enable_debug_msgs.....	363
2.10.4.9	host_if_disable_debug_msgs.....	364
2.10.4.10	host_if_debug_main_menu	364
2.10.4.11	host_if_debug_menu.....	365
2.10.4.12	host_if_debug_tick.....	366
2.10.4.13	host_if_debug.....	367
2.10.5	host_mpv_if.c.....	368
2.10.5.1	open_mpv_interface.....	369
2.10.5.2	exchange_mpv_data.....	369
2.10.5.3	exchange_mpv_data_sim.....	370
2.10.5.4	init_mpv_interface.....	370
2.10.6	host_scsi_if.c.....	371
2.10.6.1	open_scsi_interface	371
2.10.6.2	exchange_scsi_data	372

2.10.6.3	exchange_scsi_data_sim.....	373
2.10.6.4	init_scsi_interface	373
2.10.7	host_socket_if.c.....	374
2.10.7.1	open_socket_interface	374
2.10.7.2	exchange_socket_data	375
2.10.7.3	exchange_socket_data_sim.....	376
2.10.7.4	init_socket_interface	376
2.11	MPV INTERFACE (/CIG/LIBSRC/LIBMPVIDEO).....	378
2.11.1	bootforce.c.....	380
2.11.1.1	bootforce	380
2.11.1.2	set_entry_pt.....	381
2.11.1.3	rtn_entry_pt.....	382
2.11.1.4	mpvmsg_query_buf_addr.....	382
2.11.1.5	mpvmsg_reply_buf_addr.....	383
2.11.1.6	mpvmsg_to_buf_addr	383
2.11.1.7	mpvmsg_from_buf_addr.....	384
2.11.2	bootmpv.c.....	384
2.11.2.1	mpvideo_boot.....	385
2.11.2.2	prtmsgerr.....	386
2.11.2.3	prtackerr.....	387
2.11.2.4	prtstaterr.....	387
2.11.3	loadmpv.c (mpvideo_load)	388
2.11.4	mpvideo_laser.c (mpvideo_laser_request_range)	390
2.11.5	mpvideo_lut.c (mpvideo_set_lut)	391
2.11.6	mpvideo_man.c	392
2.11.6.1	mpvideo_set_video.....	392
2.11.6.2	mpvideo_setup.....	393
2.11.6.3	mpvideo_stop.....	393
2.11.6.4	mpvideo_sim_init	394
2.11.6.5	mpvideo_send_req	395
2.11.6.6	mpvideo_get_object_addr	395
2.11.7	mpvideo_mode.c.....	396
2.11.7.1	mpvideo_set_mode.....	396
2.11.7.2	mpvideo_define_mode.....	397
2.11.8	mpvideo_pass_back.c	398
2.11.9	mpvideo_pass_on.c	399
2.11.10	mpvideo_print.c.....	400
2.11.10.1	status_mpvideo_print	400

2.11.10.2	toggle_mpvideo_print.....	400
2.11.11	mpvideo_query.c (mpvideo_num_paths)	401
2.11.12	mpvideo_response.c	401
2.12	MESSAGE PROCESSING (/CIG/LIBSRC/LIBMSG).....	404
2.12.1	loc_ter_msg.c	405
2.12.2	msg_calibration_image.c	406
2.12.3	msg_cig_ctl.c	407
2.12.4	msg_dr11.c (msg_dr11_pkt_size)	408
2.12.5	msg_effect.c (msg_show_effect)	409
2.12.6	msg_end.c	410
2.12.6.1	msg_end.....	410
2.12.6.2	_downcount_effects.....	412
2.12.6.3	_display_lights.....	413
2.12.6.4	_move_load_module_stp_to_quad_buffer.....	413
2.12.6.5	_update_second_active_area_memory.....	414
2.12.6.6	_pend_on_frame_interrupt.....	414
2.12.6.7	_process_agl.....	415
2.12.6.8	_set_up_for_next_frame.....	416
2.12.6.9	_handle_request_local_terrain	416
2.12.6.10	_database_disable	417
2.12.6.11	_handle_point_lights.....	418
2.12.6.12	_reset_model_pointers.....	418
2.12.6.13	_copy_reconfigurable_viewports_section.....	419
2.12.7	msg_laser.c (msg_laser_request_range)	420
2.12.8	msg_laser_return.c	420
2.12.9	msg_lt_state.c (lt_state)	421
2.12.10	msg_pass_back.c	422
2.12.11	msg_pass_on.c	422
2.12.12	msg_ppm.c	423
2.12.12.1	msg_ppm_display_mode.....	423
2.12.12.2	msg_ppm_display_offset.....	424
2.12.12.3	msg_ppm_pixel_location.....	425
2.12.12.4	msg_ppm_pixel_state.....	425
2.12.13	msg_process_round48.c	426
2.12.14	msg_subsys_mode.c	426
2.12.15	msg_syserr.c	427
2.12.16	msg_veh_state.c.....	428
2.12.16.1	msg_otherveh_state	428
2.12.16.2	msg_staticveh_state	429

2.12.16.3	msg_staticveh_rem.....	429
2.12.17	msg_vflags.c (msg_view_flags)	430
2.12.18	msg_vport.c	431
2.12.18.1	msg_viewport_update	431
2.12.18.2	msg_view_magnification.....	432
2.12.18.3	msg_rot2x1_matrix	432
2.12.18.4	msg_rts4x3_matrix.....	433
2.12.18.5	msg_hprxyzs_matrix.....	433
2.12.18.6	msg_translation.....	433
2.12.18.7	msg_scale	434
2.12.18.8	msg_1rotation.....	434
2.12.18.9	msg_3rotations	435
2.12.19	print_msg.c	435
2.12.19.1	print_msg_*.....	436
2.12.19.2	init_print_msg_array.....	440
2.12.20	show_effect_msg.c	441
2.13	REAL-TIME PROCESSING (/CIG/LIBSRC/LIBRTT).....	442
2.13.1	agpt_init.c	444
2.13.2	bal_get_db_pos.c	444
2.13.3	bal_get_lm_grid.c	445
2.13.4	bal_routines.c.....	446
2.13.4.1	sim_bal_init.....	446
2.13.4.2	sim_bal_start.....	446
2.13.4.3	sim_bal_frame_rate	447
2.13.4.4	sim_bal_req_pt_info	447
2.13.4.5	sim_bal_agl_wanted	448
2.13.4.6	sim_bal_process_msg	449
2.13.4.7	sim_bal_process_tracer.....	450
2.13.4.8	sim_bal_traj_chord.....	451
2.13.4.9	sim_bal_round_fired.....	452
2.13.4.10	sim_bal_static_add	452
2.13.4.11	sim_bal_static_rem.....	453
2.13.4.12	sim_bal_reset	453
2.13.4.13	sim_bal_tf_veh_update	454
2.13.5	ball_effect_add.c	455
2.13.6	cal.c	455
2.13.7	cigsimio_obj.c.....	457
2.13.7.1	gos_cigsimio.....	458

2.13.7.2	cigsimio_msg_in	459
2.13.7.3	cigsimio_msg_out.....	459
2.13.7.4	cigsimio_write	460
2.13.7.5	cigsimio_get_data	461
2.13.7.6	cigsimio_buffer_init.....	462
2.13.7.7	cigsimio_frame_end.....	462
2.13.8	close_db.c	463
2.13.9	clouds.c	463
2.13.9.1	cloud_init.....	463
2.13.9.2	cloud_update.....	464
2.13.9.3	cloud_update_model	465
2.13.9.4	cloud_mgmt.....	465
2.13.9.5	cloud_placement.....	466
2.13.9.6	cloud_scud	467
2.13.9.7	set_lut.....	467
2.13.9.8	rnd.....	468
2.13.10	config_ballistics.c.....	468
2.13.10.1	config_ballistics	468
2.13.10.2	bal_buffer_setup.....	470
2.13.11	config_color_table.c	470
2.13.12	config_database.c.....	471
2.13.12.1	config_database.....	471
2.13.12.2	errors.....	473
2.13.12.3	func_msg.....	473
2.13.12.4	print_pdbase	474
2.13.13	db_mcc_setup.c	474
2.13.14	debug_initdr.c	478
2.13.15	ded_gm_pool.c.....	479
2.13.15.1	pool_init.....	479
2.13.15.2	pool_get_off_24.....	480
2.13.15.3	addr_in_pool.....	480
2.13.16	ded_model_trace.c	481
2.13.17	dl_man.c (dl_setup)	482
2.13.18	download_bvols.c	482
2.13.19	effect_downcount.c	483
2.13.20	file_control.c	484
2.13.21	fxbvtofl.c.....	486
2.13.21.1	fxbvtofl_dart.....	486

2.13.21.2	fxbvtofl	487
2.13.21.3	fxbvtofl_020	487
2.13.22	generic_lm.c	488
2.13.22.1	init_generic_lm	488
2.13.22.2	generic_lm	488
2.13.23	get_tx_lut_index.c	489
2.13.24	global_init.c	489
2.13.24.1	host_if_buffer_init	490
2.13.24.2	host_init_packet_sizes	490
2.13.25	gun_overlays.c	490
2.13.25.1	m1_gun_overlay	491
2.13.25.2	m2_gun_overlay	491
2.13.25.3	make_m1_overlays	492
2.13.25.4	make_m2_overlays	493
2.13.26	hw_test.c	494
2.13.27	init_sim.c (init_simulation)	495
2.13.28	load_dbase.c	497
2.13.29	load_esifa.c	498
2.13.30	load_modules.c	498
2.13.30.1	load_modules	499
2.13.30.2	getlmdp	499
2.13.30.3	getside	500
2.13.30.4	whatdirptr	501
2.13.31	loc_ter.c	502
2.13.31.1	local_terrain	502
2.13.31.2	local_terrain_cleanup	503
2.13.32	make_bbn.c	504
2.13.32.1	prt_mtx	504
2.13.32.2	rotate_x	505
2.13.32.3	rotate_y	505
2.13.32.4	rotate_z	506
2.13.32.5	multmatrix	506
2.13.32.6	id_matrix	507
2.13.33	mkmtx_nt.c	507
2.13.33.1	make_p_nt	508
2.13.33.2	rotate_x_nt	508
2.13.33.3	rotate_y_nt	509
2.13.33.4	rotate_z_nt	509

2.13.33.5	swap_axis.....	510
2.13.33.6	id_4x3mtx.....	510
2.13.33.7	scale_mtx.....	511
2.13.33.8	translate.....	512
2.13.33.9	mult_4x3mtx.....	512
2.13.33.10	getmatrix.....	513
2.13.33.11	matrix2.....	513
2.13.33.12	mtxcpy.....	514
2.13.33.13	mat_copy.....	514
2.13.33.14	mat_vec_mul.....	515
2.13.33.15	mat_adjugate.....	515
2.13.33.16	mat_transpose.....	516
2.13.33.17	mat_scale.....	516
2.13.33.18	mat_determinant.....	517
2.13.33.19	mat_inverse.....	517
2.13.33.20	vec_mat_mul.....	518
2.13.33.21	make4x3.....	518
2.13.34	model_mtx.c.....	519
2.13.35	mx2_hword.c.....	519
2.13.35.1	mx2_open.....	520
2.13.35.2	mx2_push.....	520
2.13.35.3	mx2_peek.....	521
2.13.35.4	mx2_skip.....	522
2.13.35.5	mx2_error.....	522
2.13.35.6	mx2_hwcopy.....	523
2.13.36	open_dbase.c.....	523
2.13.36.1	open_dbase.....	523
2.13.36.2	func_msg.....	525
2.13.37	open_ded.c.....	525
2.13.38	otherveh_state.c.....	527
2.13.39	pretend_veh.c.....	528
2.13.40	rowcol_rd.c.....	529
2.13.40.1	rowcol_rd_1.....	530
2.13.40.2	rowcol_rd_2.....	531
2.13.40.3	rowcol_rd_3.....	531
2.13.40.4	rowcol_rd_4.....	531
2.13.40.5	_rowcol_rd.....	532

2.13.40.6	rowcol_rd_1_cleanup.....	533
2.13.40.7	rowcol_rd_2_cleanup.....	533
2.13.40.8	rowcol_rd_3_cleanup.....	534
2.13.40.9	rowcol_rd_4_cleanup.....	534
2.13.40.10	_rowcol_rd_cleanup	535
2.13.41	rt_mailbox.c	535
2.13.41.1	rt_pend.....	535
2.13.41.2	rt_post	536
2.13.41.3	rt_accept.....	537
2.13.42	rtt_init.c	537
2.13.42.1	initialize_defaults	537
2.13.42.2	initialize.....	538
2.13.43	simulation.c.....	540
2.13.43.1	simulation	540
2.13.43.2	process_a_msg	541
2.13.43.3	syserr	545
2.13.44	staticveh_remove.c	546
2.13.45	staticveh_state.c	547
2.13.46	sys_control.c	548
2.13.46.1	sys_control_init.....	548
2.13.46.2	sys_frame_rate.....	549
2.13.46.3	set_leds.....	550
2.13.46.4	sys_slave_sync	550
2.13.46.5	sys_master_sync	551
2.13.46.6	get_dtp_bank.....	551
2.13.46.7	read_evc_control	552
2.13.46.8	write_evc_control	552
2.13.46.9	write_evc_frame.....	553
2.13.47	upstart.c	553
2.13.47.1	upstart.....	553
2.13.47.2	upstart_cleanup	555
2.14	SERIAL DEVICE INPUT/OUTPUT (/CIG/LIBSRC/LIBSIO).....	556
2.14.1	sio.c.....	557
2.14.1.1	sio_init.....	557
2.14.1.2	sio_write.....	558
2.14.1.3	sio_close.....	559
2.14.1.4	sio_tick	560

2.15	TOKEN PROCESSING (/CIG/LIBSRC/LIBTOKEN).....	562
2.15.1	lex.c.....	563
2.15.1.1	set_token_file.....	563
2.15.1.2	next_char.....	564
2.15.1.3	scan_number.....	564
2.15.1.4	next_token.....	565
2.15.1.5	next_line.....	565
2.15.1.6	swallow_token.....	566
2.15.1.7	get_next_token.....	566
2.15.1.8	get_current_token.....	567
2.15.1.9	get_number_value.....	567
2.15.1.10	get_string_value.....	567
2.15.2	subsys_cfg_parse.c.....	568
2.15.2.1	init_subsys_parser.....	568
2.15.2.2	parse_subsys_file.....	569
2.15.2.3	get_database_name.....	570
2.15.2.4	get_ded_name.....	570
2.15.2.5	get_final_lut_name.....	570
2.15.2.6	get_3d_lut_name.....	571
2.15.2.7	get_data_2d_name.....	571
2.15.2.8	get_esifa_name.....	572
2.15.2.9	get_color_config_name.....	572
2.15.2.10	set_database_name.....	572
2.15.2.11	set_ded_name.....	573
2.15.2.12	set_final_lut_name.....	573
2.15.2.13	set_3d_lut_name.....	574
2.15.2.14	set_data_2d_name.....	574
2.15.2.15	set_esifa_name.....	575
2.15.2.16	set_color_config_name.....	575
2.16	SYSTEM UTILITIES (/CIG/LIBSRC/LIBUTIL).....	577
2.16.1	blcopy.c.....	577
2.16.2	directory.c.....	578
2.16.2.1	OpenDir.....	578
2.16.2.2	ReadDir.....	579
2.16.2.3	CloseDir.....	579
2.16.2.4	GetFileName.....	579

2.16.3	find_field.c (FindField)	580
2.16.4	find_gtfn.c (find_fn)	581
2.16.5	sload.c	582
2.16.5.1	sload	582
2.16.5.2	get_record	583
2.16.5.3	send_data	584
2.16.5.4	check_sum	584
2.16.5.5	get_binary_data	585
2.16.5.6	get_char	585
2.16.5.7	ctoi	586
2.16.6	stdopen.c	586
2.16.7	vt100.c	587
2.16.7.1	cup	587
2.16.7.2	sgr	588
2.16.7.3	double_top	588
2.16.7.4	double_bot	589
2.16.7.5	double_off	589
2.16.7.6	blank	590
2.16.7.7	save_cur	590
2.16.7.8	restore_cur	591
2.16.7.9	scroll_reg	591
2.17	VIEWPORT CONFIGURATION (/CIG/LIBSRC/LIBVPT)	593
2.17.1	be_stubs.c	597
2.17.1.1	be_query_num_paths	597
2.17.1.2	find_be_id	598
2.17.1.3	be_query_buffer_offset	599
2.17.1.4	be_query_db0	599
2.17.1.5	be_qulm	600
2.17.1.6	be_query_lm_per_lmb_side	600
2.17.1.7	vpt_init_mode_on	601
2.17.1.8	vpt_init_mode_off	601
2.17.1.9	aam_free	602
2.17.2	cnode_child.c	602
2.17.2.1	vpt_cnode_set_bchild	602
2.17.2.2	vpt_cnode_set_stdchild	603
2.17.3	cnode_get.c (vpt_cnode_get)	603
2.17.4	cnode_process.c (vpt_cnode_process)	604

2.17.5	cnode_query.c.....	606
2.17.5.1	vpt_cnode_query	606
2.17.5.2	vpt_cnode_qroot.....	607
2.17.6	cnode_set.c	607
2.17.6.1	vpt_cnode_set_branch	607
2.17.6.2	vpt_cnode_set_matrix.....	608
2.17.7	flagoff.c	609
2.17.8	globs.c (vpti_*)	610
2.17.9	init_free.c	614
2.17.9.1	vpt_root_init	614
2.17.9.2	vpt_tree_init.....	615
2.17.9.3	vpt_tree_free	616
2.17.10	linkvpt.c (vpt_cnode_linkvpt)	617
2.17.11	mtx_concat.c (concat_mtx)	618
2.17.12	mtx_dump.c	619
2.17.12.1	r4mat_dump.....	619
2.17.12.2	r8mat_dump.....	620
2.17.13	mtx_viewspace.c.....	620
2.17.13.1	mtx_non_perspective	620
2.17.13.2	mtx_perspective	621
2.17.13.3	old_mtx_perspective	622
2.17.14	path.c	622
2.17.14.1	vpt_path_process.....	623
2.17.14.2	vpt_path_update.....	624
2.17.15	path_init.c.....	625
2.17.15.1	vpt_path_get	625
2.17.15.2	vpt_path_init.....	626
2.17.15.3	vpt_path_setup.....	626
2.17.16	path_query.c (vpt_path_query)	627
2.17.17	trav_tree.c	628
2.17.18	tst_edebug.c	628
2.17.19	tst_equery.c.....	630
2.17.19.1	tst_equery	630
2.17.19.2	vptq_grptrs.....	631
2.17.19.3	vptq_vpptrs	632
2.17.19.4	vptq_cnptrs.....	632
2.17.19.5	vptq_brvals.....	633
2.17.19.6	vptq_activept.....	633

2.17.19.7	vptq_vptbrout.....	634
2.17.19.8	vptq_dynmtx.....	634
2.17.19.9	vptq_cnout.....	635
2.17.19.10	vptq_vpout.....	635
2.17.19.11	vptq_grout.....	636
2.17.20	tst_ereadconfig.c.....	636
2.17.20.1	tst_ereadconfig.....	637
2.17.20.2	p_configtree_node.....	637
2.17.20.3	p_viewport_state.....	638
2.17.20.4	p_overlay_setup.....	639
2.17.20.5	setup_p_terrain_feedback.....	639
2.17.20.6	p_terrain_feedback.....	640
2.17.21	tst_eupdate.c.....	640
2.17.22	tst_tree.c.....	642
2.17.22.1	tst_tree.....	642
2.17.22.2	mem_check.....	643
2.17.23	tst_treetrace.c.....	644
2.17.23.1	tst_treetrace.....	644
2.17.23.2	pr_branch.....	645
2.17.23.3	pr_matrix.....	645
2.17.24	u_brmask.c (vpt_update_brmask).....	646
2.17.25	u_path.c (vpt_update_one_path).....	646
2.17.26	u_rotations.c.....	647
2.17.26.1	vpt_update_2x1_heading.....	648
2.17.26.2	vpt_update_2x1_pitch.....	648
2.17.26.3	vpt_update_2x1_roll.....	649
2.17.26.4	vpt_update_heading.....	650
2.17.26.5	vpt_update_pitch.....	650
2.17.26.6	vpt_update_roll.....	651
2.17.26.7	vpt_update_hpr.....	651
2.17.27	u_viewport.c.....	652
2.17.27.1	vpt_update_fov.....	652
2.17.27.2	vpt_update_fov_lod.....	653
2.17.27.3	vpt_update_lodm.....	654
2.17.27.4	vpt_update_near_plane.....	654
2.17.27.5	vpt_update_rez.....	655
2.17.27.6	vpt_update_view_range.....	655

2.17.27.7	vpt_update_all	656
2.17.28	u_xfrm.c	657
2.17.28.1	vpt_update_4x3_matrix.....	657
2.17.28.2	vpt_update_3x3_matrix.....	658
2.17.28.3	vpt_update_hprxyzs.....	658
2.17.28.4	vpt_update_scale	659
2.17.28.5	vpt_update_translation.....	660
2.17.29	update_mtx.c (vpt_update_mtx)	660
2.17.30	update_rot.c (vpt_update_rot)	661
2.17.31	vpt_get.c (vpt_vpt_get)	662
2.17.32	vpt_process.c (vpt_vpt_process)	663
2.17.33	vpt_query.c (vpt_vpt_query)	664
2.17.34	vpt_set.c (vpt_vpt_set)	665
2.17.35	vpt_update.c	666
2.18	FORCE PROCESSING (/CIG/OTHERSRC/FORCE).....	668
2.18.1	data_type.c	674
2.18.2	exception.asm	675
2.18.2.1	excep_init.....	675
2.18.2.2	spur_int.....	675
2.18.3	f0_3dlut_download.c	676
2.18.4	f0_3dlut_switch.c	677
2.18.5	f0_alllut_switch.c	678
2.18.6	f0_debug_disable.c	679
2.18.7	f0_debug_enable.c	679
2.18.8	f0_final_lut_download.c	680
2.18.9	f0_final_lut_switch.c	681
2.18.10	f0_mode_select.c	682
2.18.11	f0_mpv_init.c	683
2.18.12	f0_mpv_lut_type_request.c	683
2.18.13	f0_mpv_peek.c	684
2.18.14	f0_mpv_poke.c	685
2.18.15	f0_mpv_poke16.c	686
2.18.16	f0_mpv_reset.c	686
2.18.17	f0_mpv_task_control.c	687
2.18.18	f0_mpv_test.c	688
2.18.19	f0_mpv_write.c	688
2.18.20	f0_pass_on.c	689
2.18.21	f0_pixel_depth_request.c	690
2.18.22	f0_query.c	691
2.18.22.1	f0_query.....	691
2.18.22.2	strcpy	692

2.18.22.3	strcat	692
2.18.22.4	strlen	693
2.18.23	f0_set_display.c	693
2.18.24	f0_trigger.c	694
2.18.25	f0_unknown.c	695
2.18.26	f1_force_init.c	696
2.18.27	f1_init_jump_table.c	697
2.18.28	f1_pixel_address.c	697
2.18.28.1	f1_pa_init	698
2.18.28.2	f1_pa_new_orientation	698
2.18.28.3	f1_pa_new_resolution	699
2.18.28.4	f1_pa_fb_offset	700
2.18.28.5	f1_pa_640x480_v	700
2.18.28.6	f1_pa_320x240_v	701
2.18.28.7	f1_pa_640x256_v	702
2.18.28.8	f1_pa_640x240_v	702
2.18.28.9	f1_pa_640x480_h	703
2.18.28.10	f1_pa_320x240_h	704
2.18.28.11	f1_pa_640x256_h	704
2.18.28.12	f1_pa_640x240_h	705
2.18.29	f1_process_messages.c	706
2.18.30	f1_setup_environment.c	707
2.18.31	force.asm	707
2.18.31.1	init_ports	708
2.18.31.2	gsp_write	708
2.18.31.3	gsp_read	709
2.18.31.4	gsp_ioctl_read	710
2.18.31.5	gsp_ioctl_write	710
2.18.31.6	gsp_reset	711
2.18.32	forcetask.c	711
2.18.32.1	main	711
2.18.32.2	compare_buffers	713
2.18.32.3	restart_clock	714
2.18.32.4	red_clock	714
2.18.32.5	se_clock	714
2.18.33	gsp_io.c	715
2.18.34	mx2_hword.c	716
2.18.34.1	mx2_open	716

2.18.34.2	mx2_push	717
2.18.34.3	mx2_peek	718
2.18.34.4	mx2_skip	718
2.18.34.5	mx2_error	719
2.18.34.6	mx2_hwcopy	719
2.18.35	nmi_type.c	720
2.18.36	poll_ready.c	721
2.18.37	test_gsp.c	722
3	RESOURCE UTILIZATION	723
3.1	DISK SPACE REQUIREMENTS	723
3.2	MEMORY REQUIREMENTS	723
APPENDIX A: SYSTEM INCLUDE FILES		724
A.1	/CIG/INCLUDE/BACKEND.H	724
A.2	/CIG/INCLUDE/BAL_REAL_TIME.H	724
A.3	/CIG/INCLUDE/BAL_STRUCT.H	724
A.4	/CIG/INCLUDE/BALLISTICS.H	724
A.5	/CIG/INCLUDE/BBNCTYPE.H	725
A.6	/CIG/INCLUDE/BM_FUNCTIONS.H	725
A.7	/CIG/INCLUDE/BP_FUNCTIONS.H	725
A.8	/CIG/INCLUDE/BX_DEFINES.H	725
A.9	/CIG/INCLUDE/BX_EXTERNS.H	726
A.10	/CIG/INCLUDE/BX_GLOBALS.H	726
A.11	/CIG/INCLUDE/BX_MACROS.H	726
A.12	/CIG/INCLUDE/BX_MESSAGES.H	727
A.13	/CIG/INCLUDE/BX_RTDB_STRUCTS.H	727
A.14	/CIG/INCLUDE/BX_STRUCTS.H	728
A.15	/CIG/INCLUDE/CIGSIMIO.H	728
A.16	/CIG/INCLUDE/CLOUDS.H	728
A.17	/GT/INCLUDE/CLPARSE.H	729
A.18	/CIG/INCLUDE/DB_STRUCT.H	729
A.19	/CIG/INCLUDE/DEF_ALLOC.H	729
A.20	/CIG/INCLUDE/DEF_MTX_TYPE.H	729
A.21	/CIG/INCLUDE/DEFINES_2D.H	730
A.22	/CIG/INCLUDE/DEFINITIONS.H	730
A.23	/CIG/INCLUDE/DEMO_STRUCT.H	731
A.24	/CIG/INCLUDE/DGI_STDC.H	731
A.25	/CIG/INCLUDE/DGI_STDG.H	733
A.26	/CIG/INCLUDE/DIG_DEFINES.H	734
A.27	/CIG/INCLUDE/DIG_STRUCT.H	734
A.28	/CIG/INCLUDE/ECOMPILE1.H	734

A.29	/CIG/INCLUDE/EMEMORY_MAP.H.....	734
A.30	/CIG/INCLUDE/ESIFA.H.....	736
A.31	/CIG/OTHERSRC/FORCE/F1_DEFINES.H.....	737
A.32	/CIG/OTHERSRC/FORCE/F1_EXTERNS.H.....	737
A.33	/CIG/OTHERSRC/FORCE/F1_GLOBALS.H.....	737
A.34	/CIG/OTHERSRC/FORCE/FORCE_DEFINES.H.....	738
A.35	/CIG/OTHERSRC/FORCE/FORCE_ENV.H.....	738
A.36	/CIG/OTHERSRC/FORCE/FORCE_RTSW_IF.H.....	738
A.37	/CIG/OTHERSRC/FORCE/FORCE.H.ASM.....	739
A.38	/CIG/INCLUDE/FUNCTIONS.H.....	739
A.39	/CIG/INCLUDE/GLOBAL_2D.H.....	740
A.40	/CIG/INCLUDE/GLOBFIR_2D.H.....	740
A.41	/CIG/INCLUDE/IF_CIG2SIM.H.....	740
A.42	/CIG/INCLUDE/IF_CTL_ERR.H.....	741
A.43	/CIG/INCLUDE/IF_HDR_STR.H.....	742
A.44	/CIG/INCLUDE/IF_INIT.H.....	742
A.45	/CIG/INCLUDE/IF_MSG_IDS.H.....	743
A.46	/CIG/INCLUDE/IF_RVA2NET.H.....	743
A.47	/CIG/INCLUDE/IF_SIM2CIG.H.....	743
A.48	/CIG/INCLUDE/IF_TST_CTL.H.....	745
A.49	/CIG/INCLUDE/IF_VEH_EFF.H.....	745
A.50	/GT/INCLUDE/IFXVISI.H.....	745
A.51	/CIG/INCLUDE/KEYWORDS.H.....	746
A.52	/CIG/INCLUDE/KLUDGE.H.....	746
A.53	/CIG/INCLUDE/LEX.H.....	746
A.54	/CIG/INCLUDE/M2_CONFIG.H.....	746
A.55	/GT/INCLUDE/MATH.H.....	747
A.56	/CIG/INCLUDE/MBX.H.....	748
A.57	/CIG/INCLUDE/MEMORY_MAP.H.....	748
A.58	/CIG/INCLUDE/MEMORY_MAP_DEFINES.H.....	749
A.59	/CIG/INCLUDE/MODEL_STRUCT.H.....	750
A.60	/CIG/OTHERSRC/FORCE/MPV_MDEF.H.....	750
A.61	/CIG/OTHERSRC/FORCE/MPV_MEMORY_DEFINES.H.....	750
A.62	/CIG/INCLUDE/MPV_STRUCT.H.....	750
A.63	/CIG/INCLUDE/MPVIDEO.H.....	751
A.64	/CIG/INCLUDE/MPVIDEO_MSG.H.....	751
A.65	/CIG/INCLUDE/MX_DEFINES.H.....	751
A.66	/CIG/INCLUDE/MX2_DEFINES.H.....	752
A.67	/CIG/INCLUDE/OVERLAY3D_STRUCT.H.....	752
A.68	/CIG/INCLUDE/OVRLY_DEFS.H.....	752
A.69	/CIG/INCLUDE/POLY_STRUCT.H.....	753
A.70	/CIG/INCLUDE/PPM.H.....	753
A.71	/CIG/INCLUDE/PRINT_MSG_EXTERNS.H.....	753

A.72	/CIG/INCLUDE/RCINCLUDE.H.....	753
A.73	/CIG/INCLUDE/REAL_TIME.H.....	754
A.74	/CIG/INCLUDE/REAL_TIMEVPT.H.....	755
A.75	/CIG/INCLUDE/RT_DEFINITIONS.H.....	756
A.76	/CIG/INCLUDE/RT_MACROS.H.....	756
A.77	/CIG/INCLUDE/RT_TYPES.H.....	756
A.78	/CIG/OTHERSRC/FORCE/RTCDEFINES.H.....	756
A.79	/CIG/INCLUDE/RTDB_STRUCT.H.....	756
A.80	/CIG/INCLUDE/SIM_CIG_DEFINES.H.....	757
A.81	/CIG/INCLUDE/SIM_CIG_IF.H.....	758
A.82	/CIG/INCLUDE/SLAVE133_FUNCTIONS.H.....	758
A.83	/CIG/INCLUDE/STANDARD.H.....	758
A.84	/GT/INCLUDE/STDIO.H.....	759
A.85	/GT/INCLUDE/STRINGS.H.....	760
A.86	/CIG/INCLUDE/STRUCT_2D.H.....	761
A.87	/CIG/INCLUDE/STRUCTURES.H.....	761
A.88	/CIG/INCLUDE/SUBSYS_CFG_PARSE.H.....	762
A.89	/GT/INCLUDE/SYSDEFS.H.....	762
A.90	/CIG/INCLUDE/SYSDEFS2.H.....	762
A.91	/CIG/INCLUDE/TFLAT.H.....	763
A.92	/CIG/INCLUDE/TFLAT_7K.H.....	763
A.93	/CIG/INCLUDE/TFLAT_FAST.H.....	763
A.94	/CIG/INCLUDE/TFLAT_SLOW.H.....	763
A.95	/CIG/INCLUDE/TRVERSE_CMD_DEFS.H.....	763
A.96	/CIG/INCLUDE/U105MMSABOT30HZ.H.....	763
A.97	/CIG/INCLUDE/U25MMHEAT.H.....	764
A.98	/CIG/LIBSRC/LIBVPT/VPI_MSGS.H.....	764
A.99	/CIG/INCLUDE/VPI_QUERY.H.....	764
A.100	/CIG/INCLUDE/VPI_STRUCT.H.....	764
A.101	/CIG/INCLUDE/VPI_VIEWPORT.H.....	765
APPENDIX B: SYSTEM MACROS.....		767
B.1	AAM1_TO_AAM2.....	767
B.2	AAM2_ADDR.....	768
B.3	AAREAD.....	768
B.4	ABSVAL.....	768
B.5	BCOPY.....	768
B.6	CHECK_FORCE.....	769
B.7	CHECKROT.....	769
B.8	CUBE.....	770
B.9	DART_ENQUEUE.....	770
B.10	DED_BOUNDARY.....	770
B.11	DEGREE_TO_RADIAN.....	771

B.12	DELETE_ROUND.....	771
B.13	DELETE_STAT_VEH.....	771
B.14	DOWNLOAD_DATA.....	772
B.15	DTP.* (DTP MACROS)	772
B.16	DUMP_DART_BUFFER.....	777
B.17	ERRMSG	777
B.18	EXCHANGE_FLEA_DATA	777
B.19	FIND_LM.....	778
B.20	FLTOFX	778
B.21	FREE_LM_CACHE	779
B.22	FXT0881	779
B.23	FXT0FL	780
B.24	GET_CHORD_END.....	780
B.25	GET_DB_POS	780
B.26	GET_LB_FROM_LM	780
B.27	GLOB.....	781
B.28	INCR_COMPONENT.....	782
B.29	INIT_MTX.....	782
B.30	MAGSQ2D.....	783
B.31	MAGSQ3D.....	783
B.32	MALLOC.....	783
B.33	MAX.....	783
B.34	MIN.....	783
B.35	NEW_ROUND.....	784
B.36	NEW_STAT_VEH.....	784
B.37	OPEN_FLEA_DATA.....	785
B.38	OUTPUT_MESSAGE.....	785
B.39	PAGE_FORMAT	786
B.40	POLY.* (POLY PROCESSOR MACROS).....	787
B.41	POP_STACK.....	789
B.42	PRINTD4	789
B.43	PRINTD8	790
B.44	PRINTHEX4	790
B.45	PRINTHEX8	790
B.46	PUSH_STACK	791
B.47	RADIAN_TO_DEGREE	791
B.48	ROOM4LABEL	791
B.49	ROOMCHECK.....	792
B.50	SEND_TF_INFO	792
B.51	SET_OUT_BITS.....	793
B.52	SET_OUT_M2BITS.....	793
B.53	SET_PPM_DISPLAY_OFFSET.....	793
B.54	SET_PPM_PIXEL_LOCATION.....	793

B.55	SIGN	794
B.56	SQUARE.....	794
B.57	SYSERR	794
B.58	TODEG.....	794
B.59	TORAD.....	795
B.60	TORADIANS.....	796
B.61	TRIGGER_FORCE.....	796
B.62	VME_TO_VMX.....	796
B.63	WAIT_FORCE.....	797
B.64	WAIT_MPVIO	797
B.65	WAIT_MPVREPLY	797
B.66	XCLOSE.....	798
B.67	XLSEEK.....	798
B.68	XOPEN.....	799
B.69	XREAD.....	799
B.70	XWRITE.....	799
APPENDIX C: OPERATING SYSTEM ROUTINES		800
C.1	SPECIAL OS SERVICE LIBRARIES	800
C.2	TASK MANAGEMENT (SC_*) ROUTINES	801
C.3	IFX FILE MANAGEMENT (IFX_*) ROUTINES.....	802
C.4	GTOS ROUTINES	803
C.5	STANDARD C RUNTIME LIBRARIES.....	804
APPENDIX D: GLOSSARY OF TERMS AND ABBREVIATIONS.....		809
APPENDIX E: CROSS-REFERENCE TABLES		815
E.1	CSUS MAPPED TO CSCS.....	816
E.2	DATA TYPE NAMES MAPPED TO TYPEDEFS.....	824
E.3	FUNCTION NAMES MAPPED TO SOURCE FILE LOCATIONS.....	835
E.4	MACRO NAMES MAPPED TO SOURCE FILE LOCATIONS	852
INDEX BY SECTION NUMBER		Index-1

1 INTRODUCTION: GT REAL-TIME SOFTWARE HOST CSCI

This document describes the GT100 Computer Image Generation (CIG) Host CSCI, also referred to as the CIG Real-Time Software. The CIG GT Host CSCI is the executable code that resides within the CIG and provides the Simulation Host (SIM) with an interface to the graphics hardware on the CIG.

1.1 The Simulator

A Vehicle Simulator Unit, or Simulator, consists of a CIG, a Simulation Host, one or more display monitors, a user, and the user's control mechanisms. Each Simulator simulates the actions of one combat vehicle, such as a tank, in real time. Multiple Simulators can be connected via a Simulation Network. The entire simulation exercise is controlled and coordinated by the Battle Manager using the Management, Command, and Control (MCC) system computer.

After the MCC initializes a Simulator at the beginning of the exercise, the vehicle's crew directs the simulation. Each Simulator reports the position, orientation, and appearance of its simulated vehicle to the MCC and the other Simulators via the network.

Figure 1-1 illustrates the relationship between the CIG, the Simulation Host, and the MCC.

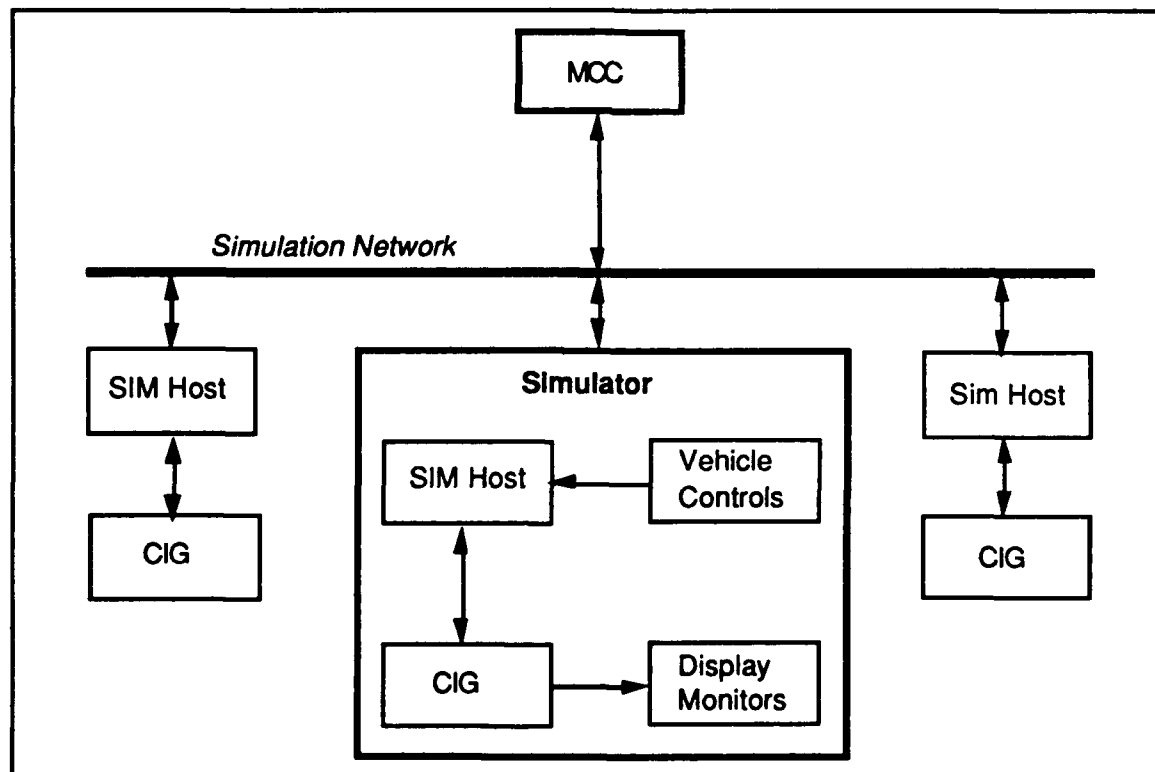


Figure 1-1. The Vehicle Simulator Unit (Simulator)

1.1.1 The Simulation Host

The Simulation Host receives and processes data from the simulation vehicle's mechanical controls, interfaces with the CIG, and communicates over the simulation network with other Simulators.

1.1.2 The CIG

The CIG interfaces with the Simulation Host, controls the images in the simulation viewports (display monitors), and houses the database that describes the simulation terrain.

The CIG can contain one or two 9U graphics processor subsystems, also called backends. (In this document, the terms "backend" and "subsystem" are used interchangeably.) A backend can be either of the following:

T backend

Generates up to eight low-resolution (320 by 200 pixels) views. These views are used in M1 and M2 Simulators.

TX backend

Generates one high-resolution (640 by 480 pixels) view or two low-resolution (320 by 240 pixels) views. These views are used in Stealth Simulators.

Having two backends provides the ability to combine high- and low-resolution views, or to increase the number of views for a simulator.

The composition of the backend(s) of a particular GT100 CIG is reflected in its model number. All model numbers have the form GT1mn, where:

- 1 indicates release 1 of the GT series
- m* is the number of TX backends
- n* is the number of T backends

Therefore, the following model numbers are possible:

GT101 = 0 TX backends, 1 T backend
GT102 = 0 TX backends, 2 T backends
GT110 = 1 TX backend, 0 T backends
GT111 = 1 TX backend, 1 T backend
GT120 = 2 TX backends, 0 T backends

The GT101 is equivalent to the earlier 120T CIG. The GT110 is equivalent to the earlier 120TX CIG.

1.2 CIG-SIM Communication

The CIG and the Simulation Host communicate by exchanging message packets, each of which is a grouping of data messages. The physical interface to the Simulation Host can be any of the following:

- DR11-W
- Ethernet
- SCSI
- MPV
- Socket

For the DR11, SCSI, MPV, and Socket interfaces, the packet buffer is a fixed size (default 4096 bytes). Any unused portion of the buffer is filled with zeroes. The packet size can be changed on the command line or in a Simulation Host message.

The Ethernet communication link adheres to the IEEE 802.3 standard and operates on the data link level. Therefore, an Ethernet packet contains the normal IEEE 802.3 24-byte preamble followed by a variable-length packet buffer. Both point-to-point and broadcast modes are supported.

Message packet exchanges occur every frame. The following frame rates are currently supported:

Frame Rate	Frame Time
10 Hz	100 ms
15 Hz	66.7 ms
30 Hz	33.3 ms
60 Hz	16.7 ms

The CIG is the clock master for all synchronous message passing. Exchanges are initiated by the CIG after it detects a frame time event (an interrupt from the hardware). Both the CIG and the Simulation Host have until the next frame to process information.

Message packets sent from the CIG describe the current state of the simulation vehicle. The Simulation Host uses this information to compute and update each parameter that affects the visual displays.

Message packets sent from the Simulation Host describe the new state of the simulation vehicle and/or changes to the simulation environment. Other messages specify where to display special effects such as bomb blasts and smoke. The CIG uses this information to compute changes in the viewing displays.

The message structures used by the CIG and the Simulation Host to communicate are documented in the "BBN GT100™ CIG to Simulation Host Interface Manual."

1.3 RTSW Software Structure

The CIG Host software is a multi-state, multi-tasking software system. It progresses through its various states upon receiving appropriate commands from the Simulation Host via the CIG-SIM message interface. The states of the CIG Host software are:

- Task Initialization
- System Configuration
- Real-Time Processing

- Stand-Alone (Flea) Mode

The simulation and other support software run as individual tasks. Using intertask mailbox locations, the tasks exchange information through shared memory. The tasks share system resources as needed, based on their relative priorities.

The 18 CSCs in the GT CIG Host CSCI are the following:

- Task Initialization
- 2-D Overlay Compiler
- Backend Manager
- Ballistics Processing
- CIG Configuration
- ESIFA Interface
- Stand-Alone Host Emulator (Flea)
- DTP Command Generator
- User Interface Mode (Gossip)
- Host Interface Manager
- MPV Interface
- Message Processing
- Real-Time Processing
- Serial Device Input/Output
- Token Processing
- System Utilities
- Viewport Configuration
- Force Processing

Figure 1-2 illustrates these CSCs.

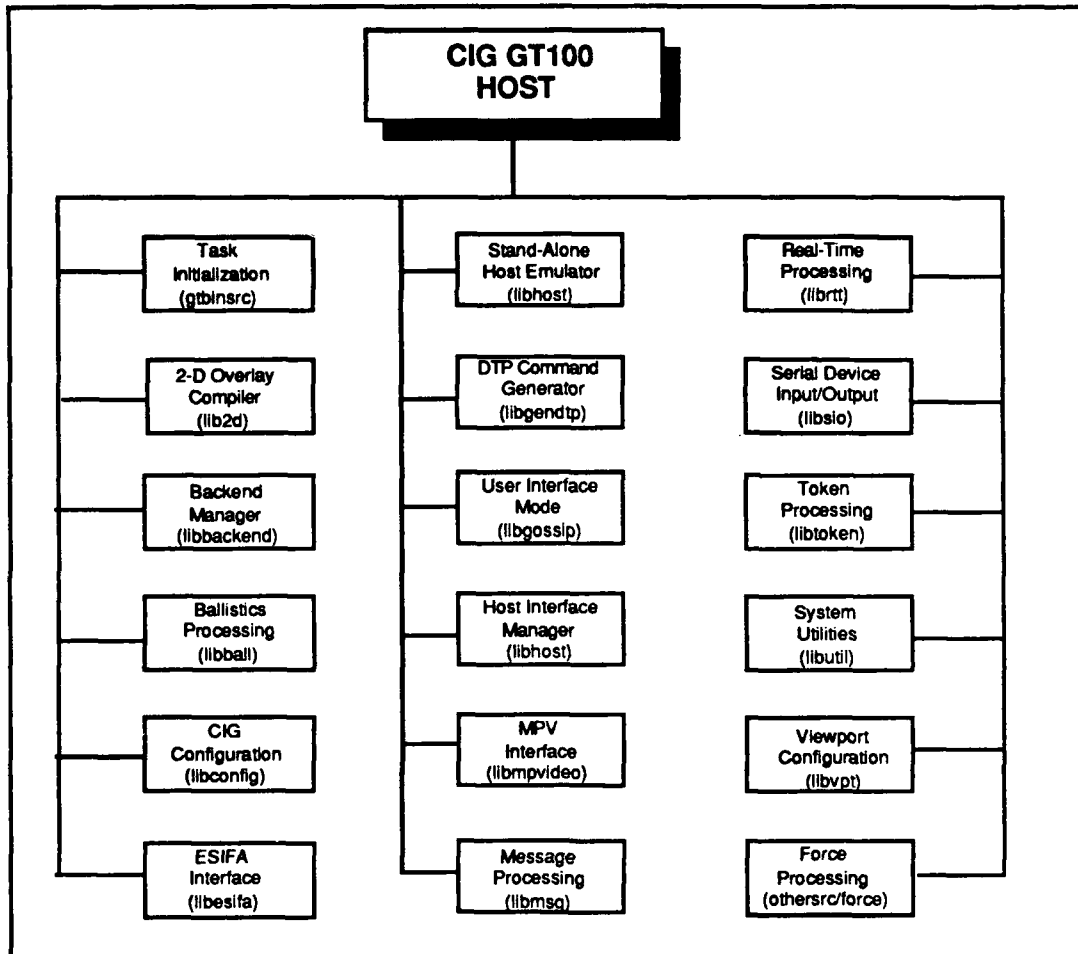


Figure 1-2. CIG GT100 Host Software CSCs

1.5 How This Document Is Organized

Section 1 (Introduction)

Provides a general overview of the CIG Real-Time Software, the Simulation Host, and the Vehicle Simulator Unit.

Section 2 (CSC Descriptions)

Describes each CSC in the CIG Real-Time Software CSCI. Each subsection begins with a general overview of the CSC, its major data structures, the primary functions it performs, and how it relates to the other CSCs. This is followed by a detailed description of each CSU in the CSC. The CSUs are presented in alphabetical order.

For the purposes of this document, a CSU is defined as a source code (.c or .asm) file. CSUs are documented as follows:

- The section heading identifies the name of the source file.

- If a CSU contains multiple functions, each is described in a separate subsection under the CSU section heading. The functions are described in the order in which they appear in the source file.
- If a CSU contains only one function, it is described under the CSU section heading. If the function name differs from the CSU name, the function name is shown in parentheses following the CSU name. If the function name matches the CSU name (minus the .c or .asm suffix), the function name is not shown in the heading.

The description of a function includes its general purpose, its function call, definitions of its parameters and return values, and a description of its processing. The description also identifies all called and calling routines.

Section 3 (Resource Utilization)

Provides disk and memory usage statistics.

Appendix A (System Include Files)

Describes the contents of each header (.h) file used in the system, and identifies the CSUs that include it. All include files are listed in alphabetical order.

Appendix B (System Macros)

Describes the macros used to perform specialized functions throughout the system, and identifies where they are used. All macros are listed in alphabetical order.

Appendix C (Operating System Service Calls)

Describes the operating system service functions and standard C libraries used by the CIG functions.

Appendix D (Glossary Of Terms And Abbreviations)

Defines some of the specialized terminology, abbreviations, and acronyms used in this document.

Appendix E (Cross-Reference Tables)

Provides lists that may help the reader locate CSUs, data type definitions, functions, and macros.

2 CSC DESCRIPTIONS

The CSCs that make up the CIG Host software system are the following:

Task Initialization (/cig/gtbinsrc)

Creates all system tasks.

2-D Overlay Compiler (/cig/libsrc/lib2d)

Generates viewport overlays for TX backends.

Backend Manager (/cig/libsrc/libbackend)

Serves as an interface between the real-time software and the backend hardware.

Ballistics Processing (/cig/libsrc/libball)

Processes fired rounds to determine if an object in the database was hit.

CIG Configuration (/cig/libsrc/libconfig)

Sets up the CIG to run a simulation.

ESIFA Interface (/cig/libsrc/libesifa)

Processes requests to and from the PPM and PPTs via the ESIFA board.

Stand-Alone Host Emulator (/cig/libsrc/libflea)

Emulates the functions of the Simulation Host to allow stand-alone testing, debugging, and demonstrations.

DTP Command Generator (/cig/libsrc/libgendtp)

Generates the commands used to drive the hardware to change the viewport displays.

User Interface Mode (/cig/libsrc/libgossip)

Provides a menu interface to system functions and memory for testing and debugging.

Host Interface Manager (/cig/libsrc/libhost)

Handles the communication between the CIG and the Simulation Host.

MPV Interface (/cig/libsrc/libmpvideo)

Processes commands to and from the Micro Processor Video (MPV) board.

Message Processing (/cig/libsrc/libmsg)

Processes many of the messages received from or returned to the Simulation Host.

Real-Time Processing (/cig/libsrc/librtt)

Drives the simulation using messages sent from the Simulation Host.

Serial Device Input/Output (/cig/libsrc/libsio)

Handles writing to and reading from a serial I/O device.

Token Processing (/cig/libsrc/libtoken)

Provides lexical functions used to parse the subsystem configuration file.

System Utilities (/cig/libsrc/libutil)

Provides general-use utilities for other CSCs.

Viewport Configuration (/cig/libsrc/libvpt)

Creates the configuration tree that describes each viewport, based on messages received from the Simulation Host.

Force Processing (/cig/othersrc/force)

Provides an interface to the MPV and the GSP via the Force board in a TX backend.

This section describes the functions performed by each of these CSCs.

2.1 Task Initialization (/cig/gtbinsrc)

The Task Initialization CSC is responsible for creating all CIG tasks. This CSC is invoked when the system is booted. It starts the other system tasks (upstart, local_terrain, flea, ballistics, etc.), then suspends itself.

The Task Initialization CSC is also responsible for processing system shutdowns. If the user asks (through Gossip) to shut down the system, the Task Initialization CSC stops all tasks and initiates the cleanup routines that deallocate the tasks' resources.

At the current time, the mkcal.c CSU, used to make calibration overlays, is also part of this CSC.

Figure 2-1 identifies the CSUs in the Task Initialization CSC.

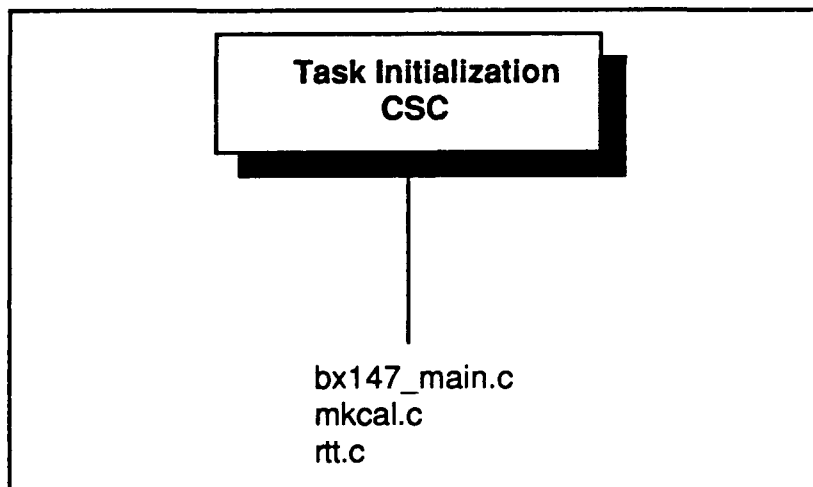


Figure 2-1. Task Initialization CSUs

2.1.1 bx147_main.c

The functions in the bx147_main.c CSU are used to invoke and shut down Ballistics. These functions are:

- bx147_main
- poll_shutdown

2.1.1.1 main

The main function in bx147_main.c is the MVME147 main Ballistics routine. main is executed via the autoboot mechanism or when the user starts the system. It spawns off the Ballistics task (bx_task) and the Ballistics diagnostic probe (bx_probe), then suspends itself.

The command line used to invoke main may include arguments that control how Ballistics is to be initialized and used.

main does the following:

- Calls clparse to parse the command line.
- Calls sc_tinquiry to determine the base task id and priority.
- Sets the base task id and priority.
- Calls pcreate to create each task's entry in the task table.
- Suspends itself.

The function exits with a 1 if clparse reports an error.

main (using pcreate) initiates the application task table in the operating system by establishing entries for the Ballistics tasks, as follows:

Entry in Task Table	Priority Offset (added to priority of main task)	Task Name	Start Flag (TRUE=start task)
bx_task	1	"bx_task"	TRUE
bx_probe	3	"bx_probe"	TRUE

Called By: none

Routines Called: clparse
exit
pcreate
printf
sc_tinquiry
sc_tsuspend
serror

Parameters: int argc
char *argv[]

Returns: none

2.1.1.2 poll_shutdown

The poll_shutdown function is a dummy routine that has no effect. (The poll_shutdown routine in rtt.c is used to determine whether a system shutdown has been initiated.)

The function call is **poll_shutdown()**. The function always returns 1.

Called By: none

Routines Called: none

Parameters: none

Returns: 1

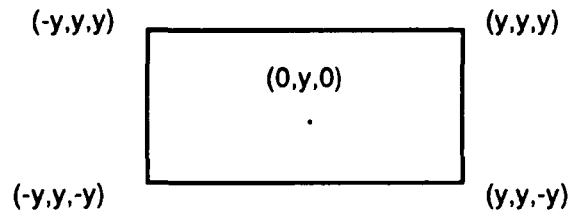
2.1.2 mkcal.c

The mkcal.c CSU contains functions used to generate the calibration overlay. This overlay is a hard-coded pattern of triangles, vertical and horizontal alignment bars, and colored rectangles. The overlay is displayed on a viewport on top of the view of the terrain. The pattern helps the Simulator user center the screen.

The functions in this CSU are:

- make_cal_overlay
- make_cal_matrices
- make_cal_patterns
- pix_mult

The Poly Processor uses perspective matrices in normalized viewspace (i.e., the field-of-view is not used) when crunching on overlay polygons. The only perspective matrix required for an overlay is a matrix to swap the axes (view space into screen space). The vertices overlay can be described to the Poly Processor as follows:



where y is the distance from the eye to the overlay.

This means that if the vertices of an overlay (such as the monitor calibration overlay) are given in pixel coordinates, they must be converted to the normalized view space coordinate system. For example, if the screen resolution is 200 x 200, a vertex with pixel coordinates (-50,100) is converted to $(-1/2,1)$.

2.1.2.1 make_cal_overlay

The make_cal_overlay function allocates memory for the calibration overlay, and calls the appropriate routines to create it. This function is called by cig_config (in the CIG Configuration CSC) as part of its initialization process.

The function call is `make_cal_overlay()`. The function does the following:

- Allocates memory for the calibration flag and overlay.
- Calls `make_cal_matrices` to generate the overlay's matrices.
- Calls `make_cal_patterns` to generate the triangles, alignment bars, and other patterns displayed on the overlay.

Called By: `cig_config`

Routines Called: `aam_malloc`
`GLOB`
`make_cal_matrices`
`make_cal_patterns`

Parameters: `none`

Returns: `none`

2.1.2.2 `make_cal_matrices`

The `make_cal_matrices` function generates matrices for the following overlays:

- The calibration overlay generated during `cig_config` initialization.
- All overlays (offset image, color image, and BBN logo) requested by the Simulation Host via the `MSG_CALIBRATION_IMAGE` message.
- All overlays (offset image, color image, and BBN logo) requested by the Gossip user via the Calibration menu.

The function call is `make_cal_matrices(pcal_dat)`, where `pcal_dat` is a pointer to the calibration overlay.

`make_cal_matrices` uses `id_4x3mtx` and `swap_axis` to generate the matrices.

Called By: `cal`
`make_cal_overlay`
`msg_calibration_image`

Routines Called: `id_4x3mtx`
`swap_axis`

Parameters: `CAL_OVRLY` `*pcal_dat`

Returns: `none`

2.1.2.3 make_cal_patterns

The `make_cal_patterns` function generates all patterns (such as triangles and rectangles) displayed on the following overlays:

- The calibration overlay generated during `cig_config` initialization.
- The offset image requested by the Simulation Host via the `MSG_CALIBRATION_IMAGE` message.
- The offset image requested by the Gossip user via the Calibration menu.

The function call is `make_cal_patterns(pcal_dat)`, where *pcal_dat* is a pointer to the calibration overlay. `make_cal_patterns` does the following:

- Outputs the calibration polygons.
- Creates the corner triangles.
- Creates the frame triangles.
- Creates the vertical and horizontal alignment bars.
- Creates the colored rectangles.
- Sets the vertices of all of the displayed patterns.

Called By: `cal`
 `make_cal_overlay`
 `msg_calibration_image`

Routines Called: `none`

Parameters: `CAL_OVRLY` **pcal_dat*

Returns: `none`

2.1.2.4 pix_mult

The `pix_mult` function converts pixel coordinates into normalized viewspace coordinates.

The function call is `pix_mult(resolution, y_dist)`, where:

resolution is the screen resolution
y_dist is the y pixel coordinate

The function divides *y_dist* by (*resolution* * .5) and returns the result as *mult*.

This function is not currently used.

Called By: `none`

Routines Called:	none	
Parameters:	INT_2 REAL_4	resolution y_dist
Returns:	mult	

2.1.3 rtt.c

The rtt.c CSU contains functions used to start up and shut down the real-time software tasks. These functions are:

- main
- check_restart
- disable_restart
- rtt_shutdown
- poll_shutdown

2.1.3.1 main

The main function is the initial task in the GT100 CIG system. main is executed from the user's terminal or via the auto-boot mechanism. main initiates the execution of all other tasks in the system, then suspends itself until a system shutdown is requested.

The command line used to start up the CIG may include arguments that control how the system is to be initialized and used. For descriptions of these arguments, see the initialize function in the Real-Time Processing CSC.

main does the following:

- Calls check_restart to see if the CIG's previous execution terminated unsuccessfully. If so, prompts the user to reset the CIG, then exits with a 0.
- Calls initialize to parse the command line and invoke the options specified by the user.
- Calls sc_tinquiry to determine the base task id and priority.
- Sets the base task id and priority.
- Calls sc_screate to set up the initiate_shutdown semaphore. (This semaphore is used to indicate that the user has requested a system shutdown.)
- Calls sc_fcreate to set up the shutdown_status flag group. (This group is used to indicate which tasks have successfully shut down. A shutdown mask is used to assign a bit in the flag group to each task.)
- Calls config_database to look at the database configuration file and determine how many rowcol_rd tasks are needed. (Each backend requires its own rowcol_rd task to bring new rows/columns of load modules from the terrain database into active area memory when needed.)
- Calls config_ballistics to configure and start Ballistics Processing.
- If AGPT mode was requested on the command line, calls agpt_init to start up the sim_server task. (AGPT mode is a non-standard version of the GT100 system and is not addressed in this document.)

- Determines how many tasks are in the task table; if more than 31, displays a warning that the system may not be able to shut down properly, then calls `disable_restart` to prevent the CIG from being restarted without a reset.
- Calls `pcreate` to create each task's entry in the task table.
- If Flea mode was requested on the command line, calls `flea_host_if` to initialize the Flea interface.
- Prompts the user to enter any character to continue.
- Calls `sc_spend` to wait to be signaled to initiate a shutdown.
- If a shutdown request is posted:
 - Displays a shutdown message.
 - Calls `rtt_shutdown` to signal all tasks to shut down.
 - Calls `sc_tinquiry` to find any tasks that are suspended.
 - Calls `sc_tresume` to wake up the suspended tasks.
 - Calls `sc_fpend` to wait for all tasks to shut down (by waiting for all bits in the `shutdown_status` flag to be set).
 - If any tasks do not terminate within the required timeout, outputs an error and calls `disable_restart` to prevent the CIG from being restarted without a reset.
 - Calls `sysrup_off` to disable system interrupts.
 - Calls `sc_sdelete` to delete the `initiate_shutdown` semaphore.
 - Calls `sc_fdelete` to delete the `shutdown_status` flag group.

`main` (using `pcreate`) initiates the application task table in the operating system by establishing entries for the GT100 tasks, as shown below. Note the following:

Priority offset

This value is added to the priority of the main task.

Start flag

This flag is TRUE if the task is to be started when loaded.

Cleanup routine

This routine is the function responsible for deallocating that task's resources when the system is shut down. The `poll_shutdown` function uses the `*task_cleanup` function pointer to invoke the correct function.

Entry in Task Table	Priority Offset	Task Name	Start Flag	Cleanup Routine
bx_task	3	"ballistics"	FALSE	bx_task_cleanup
flea	4	"flea"	TRUE	flea_cleanup
flea_io_task	6	"flea_io_task"	TRUE	flea_io_task_cleanup
gossip	7	"gossip"	TRUE	gossip_cleanup
local_terrain	2	"local_terrain"	TRUE	local_terrain_cleanup
rowcol_rd_1	5	"rowcol_rd #1"	FALSE	rowcol_rd_1_cleanup
rowcol_rd_2	5	"rowcol_rd #2"	FALSE	rowcol_rd_2_cleanup
rowcol_rd_3	5	"rowcol_rd #3"	FALSE	rowcol_rd_3_cleanup
rowcol_rd_4	5	"rowcol_rd #4"	FALSE	rowcol_rd_4_cleanup
sim_server	1	"simnet_server"	FALSE	sim_server_cleanup
upstart	1	"upstart"	TRUE	upstart_cleanup

Called By: none (initiated by the operating system on start-up)

Routines Called:

- agpt_init
- check_restart
- config_ballistics
- config_database
- disable_restart
- exit
- flea_host_if
- initialize
- pcreate
- printf
- rtt_shutdown
- sc_fcreate
- sc_fdelete
- sc_finquiry
- sc_fpend
- sc_screate
- sc_sdelete
- sc_spend
- sc_tinquiry
- sc_tresume
- seerror
- sysrup_off

Parameters: int argc
char *argv[]

Returns: none

2.1.3.2 check_restart

The `check_restart` function checks to see if the system terminated abnormally from the last session. If so, the user is told to reset the CIG before restarting.

The function call is `check_restart()`.

The function calls `mpv_nfind` to get the status of the "cigran" program. The `disable_restart` function sets the status of *cigran* to 0 if restart is to be prevented due to abnormal termination of the previous session. If `check_restart` sees that the status of *cigran* is 0, it returns a 1 to main, indicating that a restart is not allowed.

The *result* returned by the function is 0 if a restart is allowed, or 1 if restarts are disabled. Any other value returned is an error from `mpv_nfind`.

Called By: main (in rtt.c)

Routines Called: `mpv_nfind`

Parameters: none

Returns: *result*

2.1.3.3 disable_restart

The `disable_restart` function sets a flag that indicates that the CIG must be rebooted before it can be restarted. This function is called if the main task determines that the previous session did not shut down properly within the allotted timeout, or that there are too many system tasks to enable an orderly shutdown. The flag set by `disable_restart` is read by `check_restart`.

The function call is `disable_restart()`.

The function calls `mpv_ncatalog` and sets the status of "cigran" to 0. This value triggers `check_restart` to return a 1 to main, indicating that no more restarts are allowed before the system is rebooted.

`disable_restart` returns the *result* returned by `mpv_ncatalog`. This is `RET_OK` if successful; any other value indicates an error.

Called By: main (in rtt.c)

Routines Called: `mpv_ncatalog`

Parameters: none

Returns: result

2.1.3.4 rtt_shutdown

The `rtt_shutdown` function initiates the shutdown procedure by posting a message to the `initiate_shutdown` semaphore. This message tells the main function that a shutdown has been requested. `rtt_shutdown` is called if the Gossip user requests a shutdown by selecting the Q ("Quit to GTOS") option on the Gossip main menu. This function is also called by main after it wakes up, causing the message to be reposted.

An overview of the shutdown procedure is as follows:

1. `rtt_shutdown` posts to the `initiate_shutdown` semaphore.
2. main, which had been pending on the `initiate_shutdown` semaphore, wakes up and immediately calls `rtt_shutdown` again, causing it to repost to the `initiate_shutdown` semaphore.
3. main activates any suspended tasks, then pends on the `shutdown_status` flag group with a finite timeout. (This flag group contains one bit for each task.)
4. Each task periodically calls `poll_shutdown` to check the `initiate_shutdown` semaphore. If `poll_shutdown` determines that the semaphore is set, it executes the calling task's cleanup function, sets the task's bit in the `shutdown_status` flag group, then terminates the task.
5. If all expected bits get set in the `shutdown_status` flag group, main outputs a message, cleans up its own resources, then exits. If an expected bit is *not* set, the pend on the flag group times out, main outputs a warning identifying the offending task(s), calls `disable_restart` to prevent a restart without a reset, cleans up its resources, then exits.

The function call is `rtt_shutdown()`.

Called By: gossip_tick
main (in rtt.c)

Routines Called: printf
sc_spost
serror

Parameters: none

Returns: none

2.1.3.5 poll_shutdown

The poll_shutdown function checks to see if a system shutdown has been initiated. This function is called periodically by each real-time task. If a shutdown has been initiated, poll_shutdown calls the task's cleanup function and terminates the task.

The function call is **poll_shutdown()**. The function does the following:

- Calls **sc_sinqury** to see if a shutdown has been initiated.
- If a shutdown has been initiated:
 - Calls **sc_tinqury** to get the calling task's id.
 - Calls the task's cleanup routine using the ***task_cleanup** function pointer. (The cleanup function for each task is specified in the task table.)
 - Calls **sc_fpost** to set the task's bit in the shutdown_status semaphore. (This indicates that the task has shut down. If all expected bits are set within the allotted time, main determines that the system shut down properly.)
 - Calls **ifx_tdelete** to delete the calling task.

The function always returns 1.

Called By: _rowcol_rd
 bx_task
 db_mcc_setup
 file_control
 flea
 flea_io_task
 gossip
 hw_test
 local_terrain
 simulation
 upstart

Routines Called: *task_cleanup
 ifx_tdelete
 printf
 sc_fpost
 sc_sinqury
 sc_tinqury
 serror

Parameters: none

Returns: 1

2.2 2-D Overlay Compiler (/cig/libsrc/lib2d)

This section describes the functions that make up the 2-D (Two-Dimensional) Overlay Compiler CSC. These functions build 2-D overlays from ASCII commands, then generate executable commands for the 2-D processor.

Note: These functions apply to TX backends only. The only overlays available on T backends are the hard-coded gun, gunner, and calibration overlays generated in the Real-Time Processing CSC.

Two-dimensional overlays are displayed on a viewport on top of the three-dimensional terrain display. For example, overlays can be used to display calibration patterns and numerical readouts such as current altitude and speed. Each 2-D component is classified as either dynamic (able to move or change) or static (not capable of movement or change).

The 2-D overlay database describes all components that can be displayed in the overlays. This database is an ASCII file sent from the Simulation Host via messages. The overall process for creating the 2-D overlay database is as follows:

1. The Simulation Host invokes the 2-D compiler using the CIG Control - Start 2D Setup message.
2. The Simulation Host sends the ASCII file via 2-D SETUP messages, one per packet buffer.
3. After the entire file has been sent, the Simulation Host sends a CIG Control - Stop message.
4. The 2-D compiler function compiles the data. If a monitor is available, error and status information is displayed.
5. The data is downloaded via the Force board into 2-D dynamic memory on the GSP (Graphics System Processor) chip on the MPV board.

Once the 2-D database is loaded into memory, the overlays can be changed using PASS_ON messages sent from the Simulation Host. These messages contain commands that are used to move or change dynamic components, and to draw or remove static components. The 2-D task (which runs on the GSP) decodes the runtime commands and updates the component information in the 2-D database accordingly. The 2-D task then processes the changes to each component in the order in which they are defined in the database.

The functions in the 2-D Overlay Compiler CSC do not process runtime changes. Update commands are passed directly from the real-time software to the GSP via an MPV Interface routine, and the GSP processes the changes to the structures in its memory.

For the complete syntax of each command used to create or change a 2-D image, refer to the "2-D Commands and Parameters" document. That document also provides a sample ASCII input file and the 2-D overlay it creates.

The 2-D overlays can also be created and compiled offline (off the CIG). Special versions of the 2-D compiler functions are used to read the overlay file and generate a binary file.

This file can then be copied to the CIG and downloaded to 2-D memory at a later time. The source files that contain the functions used to process a file offline are prefixed by `u_`. A separate "make" file is used at system build time to select these source files instead of their online equivalents.

The primary data structures built by the 2-D compiler are the following:

Component descriptor table

Contains each component's number (0-63), color, channel (0 for high resolution, 1 or 2 for low resolution), plane (foreground or background), window number (0 for screen space, 1-15 for user-defined windows), clipping values, pre-translate (pre-rotation) values, and post-translate (post-rotation) values.

Window descriptor table

Contains each window's absolute address, width (horizontal pixels), height (vertical pixels), pitch, and a conversion factor for GSP.

Component pointer table

Contains a pointer to each component in the 2-D database.

After compilation, these structures are downloaded into GSP memory via the Force board. (If the 2-D compiler is being run offline, the data is written to a binary file which can be downloaded to the GSP later.) Figure 2-2 illustrates these structures, their contents, and their interrelationships, as they exist in GSP memory.

The primitive types handled by the 2-D compiler, and the functions used to process them, are the following:

Primitive	2-D Setup Function
bit_blt	setup_bit_blt
draw_line	setup_draw_line
draw_oval	setup_oval_rectangle
draw_rect	setup_oval_rectangle
fill_oval	setup_oval_rectangle
fill_poly	setup_poly
fill_rect	setup_oval_rectangle
polyline	setup_poly
string	setup_define_string
text	setup_text

The specified function is responsible for retrieving the parameters associated with the primitive, validating the data, then adding the data to the component descriptor table.

The structure of each of these primitives is illustrated in Figure 2-3.

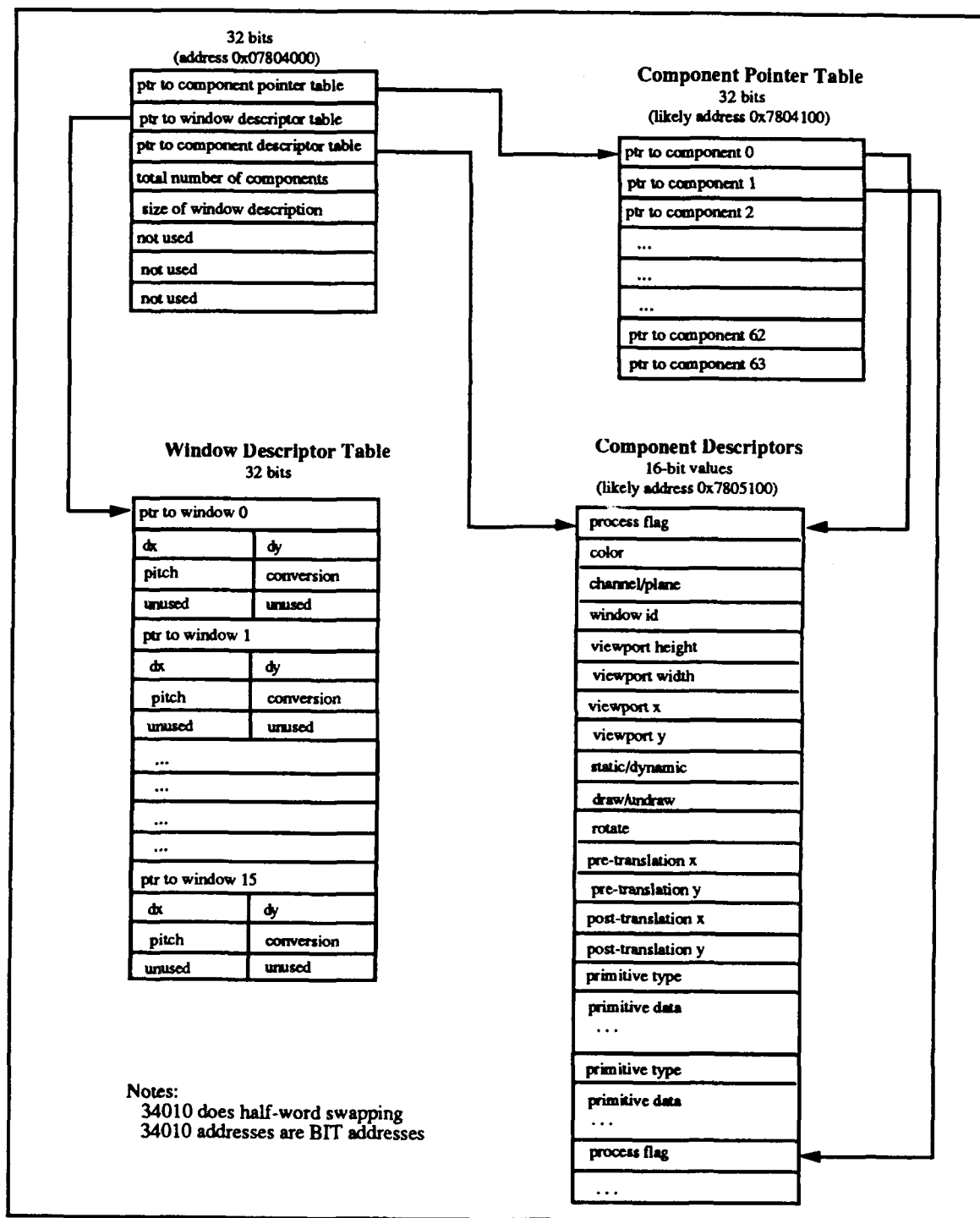


Figure 2-2. 2-D Memory (from the 2-D Compiler)

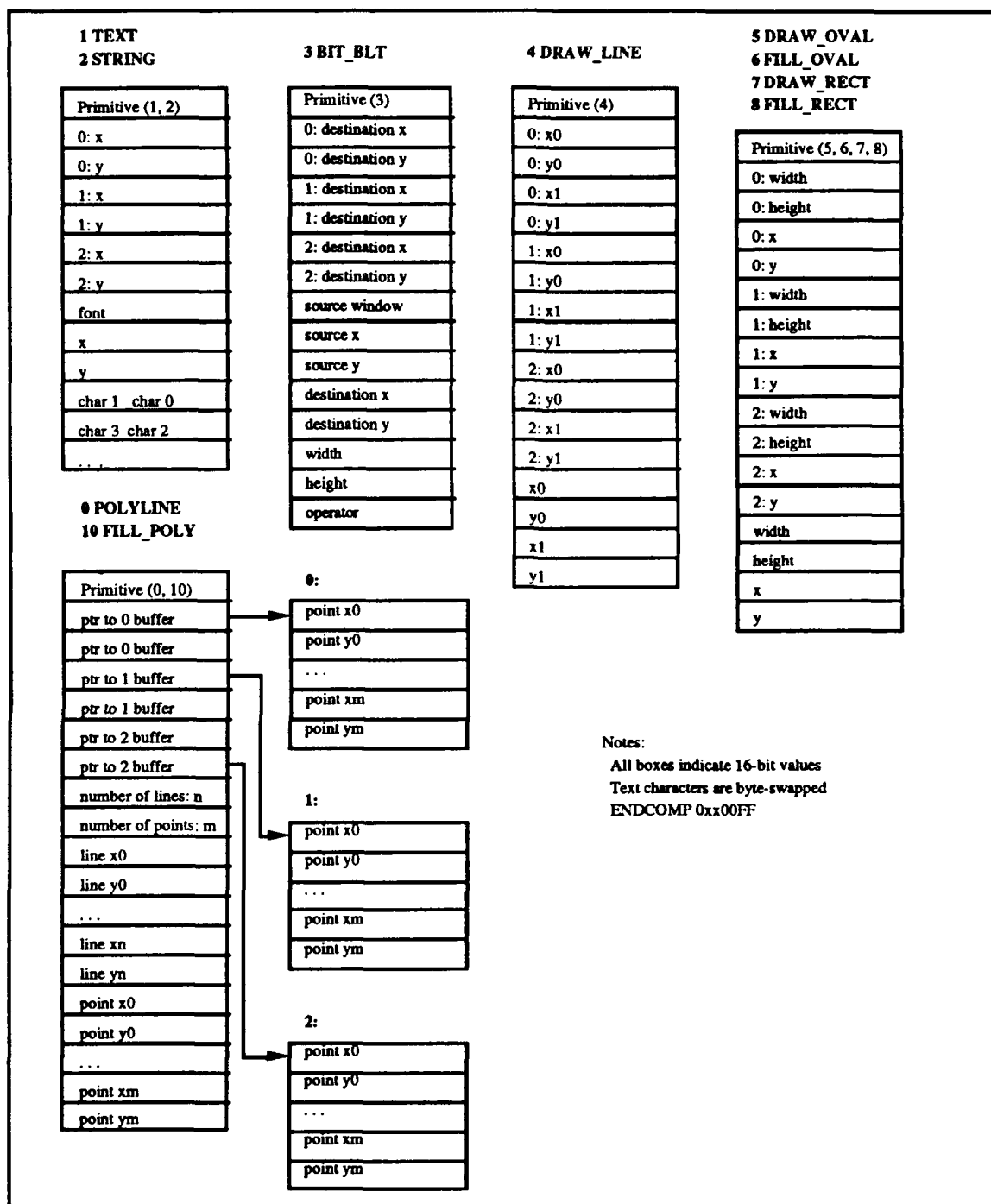


Figure 2-3. 2-D Compiler Primitives

Figure 2-4 identifies the CSUs in the 2-D Overlay Compiler CSC. These CSUs are described in this section.

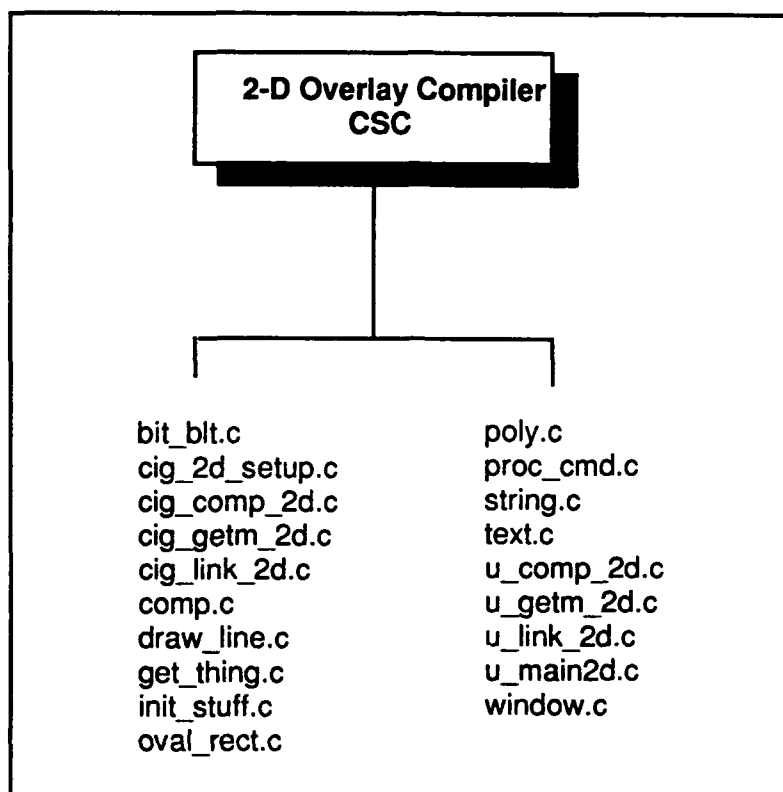


Figure 2-4. 2-D Overlay Compiler CSUs

Figure 2-5 illustrates how the CSUs in the 2-D Overlay Compiler CSC interact. This diagram illustrates the flow of events when 2-D overlays are created on the CIG from messages received from the Simulation Host. It does not reflect the process used to compile a binary file offline.

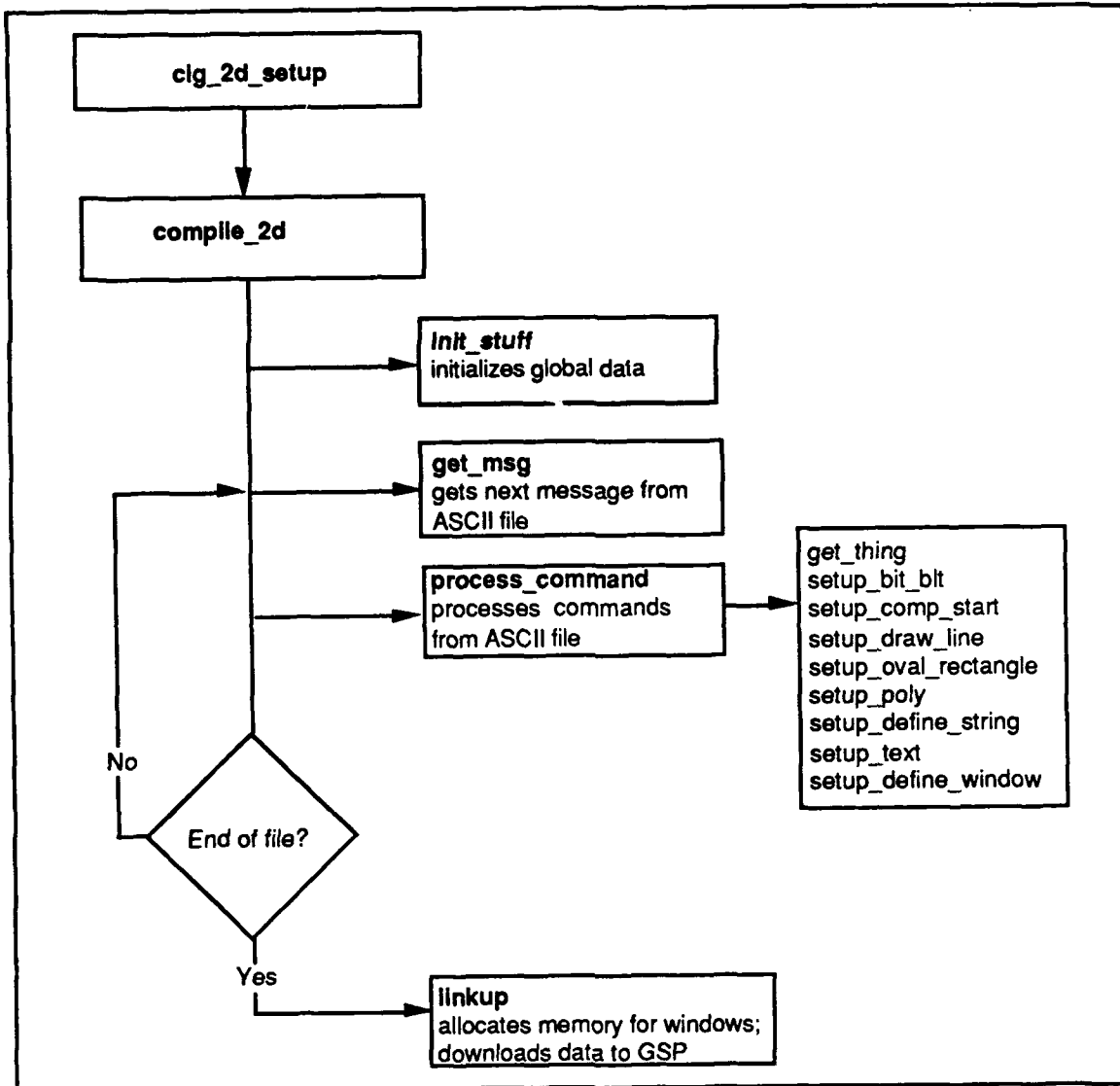


Figure 2-5. 2-D Overlay Compiler Flow Diagram

2.2.1 bit_blt.c (setup_bit_blt)

The `setup_bit_blt` function is responsible for setting up block-transferring pixel information in the component descriptor table. This function is called by `process_command` if the 2-D command to be processed is `BIT_BLT`.

The function call is `setup_bit_blt(cmd)`, where *cmd* is the command (`N_BIT_BLT`) to be processed.

`setup_bit_blt` does the following:

- Verifies that component start data has already been processed.
- Calls `get_thing` to retrieve the data in the message.
- Determines if the component descriptor table has room available.
- Places the source window pixel x and y into the component descriptor table.
- Places the destination window pixel x and y into the component descriptor table.
- Places the width, height, and operator into the component descriptor table.

If successful, the function returns a *rtn_val* of 0 (SUCCESS). If an error occurs, the *rtn_val* is one of the following:

88 (COMPONENT_DESCRIPTOR_TBL_FULL)

The table does not have enough room for the new data.

99 (SYNTAX_ERROR)

The data in the message is invalid.

Called By: `process_command`

Routines Called: `get_thing`
`printf`

Parameters: `int` `cmd`

Returns: `rtn_val`

2.2.2 cig_2d_setup.c

The `cig_2d_setup` function is the 2-D overlay setup handler. This function is called by `db_mcc_setup` if the message from the Simulation Host is `MSG_CIG_CTL - C_START_2D_SETUP`.

The function call is `cig_2d_setup(backend)`, where *backend* is the subsystem id specified by the Simulation Host.

`cig_2d_setup` does the following:

- Calls `mpvideo_get_object_addr` to get the MPV I/O board mailbox address.

- Calls `calloc` to allocate memory for the setup.
- Calls `compile_2d` to start the 2-D compiler.
- Calls `free` to deallocate the memory when the compiler is finished.

The function returns 0 if successful. It returns EOF if the MPV address returned to it is NULL, indicating that the specified backend does not contain an MPV board.

Called By: `db_mcc_setup`

Routines Called: `calloc`
`compile_2d`
`free`
`mpvideo_get_object_addr`
`printf`

Parameters: `UNS_4` `backend`

Returns: 0
EOF

2.2.3 `cig_comp_2d.c` (`compile_2d`)

The `compile_2d` function is the main driver for the 2-D database compiler. It is responsible for processing the 2-D setup messages (`MSG_2D_SETUP`) sent from the Simulation Host. Each message represents one line in the ASCII 2-D database file.

The function call is `compile_2d(pmpv)`, where *pmpv* is a pointer to the MPV object. `compile_2d` does the following:

- Sets the MPV's `override_2d` flag to FALSE.
- Calls `init_stuff` to initialize various compiler variables.
- Calls `get_msg_2d` to get the first line of the input file.
- Calls `process_command` to process each command from the input file. (`process_command` calls `get_thing` which in turn calls `get_msg_2d` to get each message from the file.)
- Checks for errors from `process_command`.
- Calls `linkup` to set up the window pointers and write the data to the GSP in the specified backend.
- Reports the number of errors detected during the compile.
- If the compilation was successful, sets the MPV's `override_2d` flag to TRUE. This tells `mpvideo_load` (in the MPV Interface CSC) to *not* load a 2-D data file specified by the Simulation Host or the `subsys.cfg` file.

Called By: `cig_2d_setup`

Routines Called: `get_msg_2d`
`init_stuff`

linkup
printf
process_command

Parameters: MPVIDEO_OBJ *pmpv

Returns: none

2.2.4 cig_getm_2d.c (get_msg_2d)

The get_msg_2d function gets the next 2-D overlay message from the input file and sets a pointer to it for the 2-D compiler.

The function call is get_msg_2d(). get_msg_2d does the following:

- Makes sure the message header size plus the message size does not exceed the packet size; calls SYSERR to generate an error message if it does.
- Calls cigsimio_msg_in to write the message to a buffer (if debug display or message recording is enabled).
- Processes each message.

The following table summarizes the steps performed by get_msg_2d to process each valid message it finds. The first column lists the messages in alphabetical order. The second column identifies the purpose of the message in italics, then lists the major steps performed by get_msg_2d.

2-D Overlay Message	Processing by get_msg_2d
MSG_2D_SETUP	<i>Provides one line of data from the ASCII input file.</i> Gets a pointer to the message for compile_2d; sets msg_code to CONTINUE_2D_SETUP.
MSG_CIG_CTL C_STOP C_NULL other	<i>Signals a state transition.</i> Sets msg_code to STOP_2D_SETUP. No action. Sets msg_code to INVALID_2D_SETUP.
MSG_END	<i>Signals the end of the message packet.</i> Calls cigsimio_msg_out to write the message to a buffer (if debug display or message recording is enabled); calls start_watch; calls the appropriate exchange_data routine (using *exchange_data) to send output and receive input buffers.

The msg_code returned by the function is one of the following:

0 (CONTINUE_2D_SETUP)	A valid 2-D setup message was found.
96 (INVALID_2D_SETUP)	An unknown message was detected.
97 (STOP_2D_SETUP)	A CIG Control-Stop message was detected.

Called By: compile_2d
 get_thing

Routines Called: *exchange_data
 cigsimio_msg_in
 cigsimio_msg_out
 printf
 start_watch
 SYSERR

Parameters: none

Returns: msg_code

2.2.5 cig_link_2d.c (linkup)

The linkup function is responsible for setting up window pointers and allocating available MPV (Micro Processor Video) memory for windows. It also downloads the 2-D overlay data to GSP memory via the MPV intertask mailbox.

The function call is **linkup(pmbx)**, where *pmbx* is a pointer to the MPV mailbox. linkup does the following:

- Verifies that the number of component start entries equals the number of component end entries.
- Calculates base addresses and table sizes for all information.
- Outputs the following data to stdout (see Figure 2-6 for a sample of the output):
 - Component pointers table base address and size.
 - Window descriptor table base address and size.
 - Component descriptor table base address and size.
 - Allocatable window base address and maximum size.
 - Base program address.
- Sets up the screen window area (this should not vary).
- Changes the component pointers to absolute addresses.
- Allocates space for the dynamic polygon buffer areas.
- Sets the allocatable window area to the space following the component descriptor table.
- Allocates space for all windows and sets the window pointers.
- Uses the DOWNLOAD_DATA macro (described in Appendix B) to download the structure information and all tables to GSP memory via the MPV mailbox and the Force control register.

Figure 2-6 is a sample of the output generated by linkup.

```

file data2d_itl.0400      - Compiler output from:
compile_2d data2d_ita.0400 data_2d_itb.0400 > data2d_itl.0400

BBN Systems and Technologies Graphics Technology Division
2D Database Compiler Date Thu Nov 17 15:23:31 PST 1988 Version: 0400
Link step starting ...
BASE COMPONENT POINTERS ADDRESS:          0x07804100
  size of component pointer table:        0x00000800
BASE WINDOW DESCRIPTOR TABLE ADDRESS:    0x07804900
  size of window descriptor table:        0x00000800
BASE COMPONENT DESCRIP TABLE ADDRESS:    0x07805100
  size of component_descriptor table:     0x000074d0
BASE ALLOCATABLE WINDOW ADDRESS:          0x0780c5e0
  maximum size of allocatable area:      0x00373a20
BASE PROGRAM ADDRESS:                    0x07b80000
  Allocating Dynamic Poly 0x3 at 0x780c5e0
  Next Available Address: 0x780ec20
  Space used: 0x2640 Space available: 0x3713e0
  Allocating Dynamic Poly 0x4 at 0x780ec20
  Next Available Address: 0x780ed40
  Space used: 0x2760 Space available: 0x3712c0
  Window 0x1 Allocated at GSP address: 0x780ed50
  Next Available Address: 0x78b6d50
  Space used: 0xaa760 Space available: 0x2c92b0
Compile finished -- Number of Errors = 0

```

Figure 2-6. Sample 2-D Compiler Output

```

Called By:      compile_2d

Routines Called:  DOWNLOAD_DATA
                  printf

Parameters:      MPVIO_INTERFACE      *pmbx

Returns:         none

```

2.2.6 comp.c (setup_comp_start)

The `setup_comp_start` function places component start data into the component descriptor table. Component start data includes the component number, color, channel number, plane (foreground, background, or none), window number, static/dynamic parameter, and rotation/translation values. This function is called by `process_command` if the 2-D command to be processed is `COMP_START`.

The function call is `setup_comp_start(cmd)`, where `cmd` is the command (`N_COMP_START`) to be processed.

`setup_comp_start` does the following:

- Calls `get_thing` to retrieve the data in the message.
- Determines if the component descriptor table has room available.
- Validates the parameters received.
- Places a pointer to the component in the component pointer table.
- Places all of the component data in the component descriptor table.

`setup_comp_start` provides some defaults if invalid parameters are encountered. The default color is white, the default plane is background, and the default static/dynamic parameter is static.

If successful, the function returns `rtn_val` set to 0 (SUCCESS). If an error is detected, the `rtn_val` is one of the following:

5 (INVALID_PARAMETERS)

One of the component parameters provided is out of range.

88 (COMPONENT_DESCRIPTOR_TBL_FULL)

The table does not have enough room for the new data.

Called By: `process_command`

Routines Called: `get_thing`
`printf`
`strcmp`

Parameters: `int` `cmd`

Returns: `rtn_val`

2.2.7 `draw_line.c` (`setup_draw_line`)

The `setup_draw_line` function puts line data into the component descriptor table. This function is called by `process_command` if the 2-D command to be processed is `DRAW_LINE`.

The function call is `setup_draw_line(cmd)`, where `cmd` is the command (`N_DRAW_LINE`) to be processed.

`setup_draw_line` does the following:

- Calls `get_thing` to retrieve the data in the message.
- Determines if the component descriptor table has room available.
- Places the line's starting point `x` (column) and `y` (row), and the ending point `x` and `y`, into the component descriptor table.

If successful, the function returns `rtn_val` set to 0 (SUCCESS). If an error is detected, the `rtn_val` is one of the following:

88 (COMPONENT_DESCRIPTOR_TBL_FULL)

The table does not have enough room for the new data.

99 (SYNTAX_ERROR)

The data in the message is invalid.

Called By: process_command

Routines Called: get_thing
printf

Parameters: int cmd

Returns: rtn_val

2.2.8 get_thing.c

The `get_thing` function scans input lines for a specified number of data items of a specified type (data, command, or text). This function is used to retrieve the data in the 2-D setup messages.

The function call is `get_thing(type, number)`, where:

type is the type of item (DATA_TYPE, COMMAND_TYPE, or TEXT_TYPE)
number is the number of items to be read

`get_thing` does the following:

- Discards blank spaces and tab characters.
- Sets a pointer to the data if any of the following is true:
 - A digit is found and *type* is DATA_TYPE.
 - An alpha character is found and *type* is COMMAND_TYPE.
 - A quote character is found and *type* is TEXT_TYPE.
- Calls `get_msg_2d` to read the next line if the end of line or comment is found.

This process continues until an error occurs or the specified number of items is read. The *rtn_val* returned by the function is 0 (SUCCESS) if the items were read successfully, or 99 (SYNTAX_ERROR) if unexpected data was found.

Called By: process_command
setup_bit_blt
setup_comp_start
setup_define_string
setup_define_window
setup_draw_line
setup_oval_rectangle
setup_poly
setup_text

Routines Called:	get_msg_2d isalpha isdigit printf	
Parameters:	int int	type number
Returns:	rtn_val	

2.2.9 init_stuff.c

The `init_stuff` function is called at the beginning of the 2-D compilation process to initialize the following global data:

- Window descriptor table.
- Allocation list.
- Component pointer table.
- Component descriptor table.

The function call is `init_stuff()`.

Called By:	compile_2d
Routines Called:	none
Parameters:	none
Returns:	none

2.2.10 oval_rect.c (setup_oval_rectangle)

The `setup_oval_rectangle` function places oval and rectangle data into the component descriptor table. This function is called by `process_command` if the 2-D command to be processed is `DRAW_OVAL`, `FILL_OVAL`, `DRAW_RECT`, or `FILL_RECT`.

The function call is `setup_oval_rectangle(cmd)`, where *cmd* is the command (`N_DRAW_OVAL`, `N_FILL_OVAL`, `N_DRAW_RECT`, or `N_FILL_RECT`) to be processed.

`setup_oval_rectangle` does the following:

- Calls `get_thing` to retrieve the data in the message.
- Determines if the component descriptor table has room available.
- Places the object's width and height into the component descriptor table.

- Places the object's x (column of the upper left corner) and y (row of the upper left corner) coordinates into the component descriptor table.

If successful, the function returns *rtn_val* set to 0 (SUCCESS). If an error is detected, the *rtn_val* is one of the following:

- 88 (COMPONENT_DESCRIPTOR_TBL_FULL)
The table does not have enough room for the new data.
- 99 (SYNTAX_ERROR)
The data could not be processed.

Called By: process_command

Routines Called: get_thing
printf

Parameters: int cmd

Returns: rtn_val

2.2.11 poly.c (setup_poly)

The setup_poly function is responsible for updating polygon data in the component descriptor table. This function is called by process_command if the 2-D command to be processed is POLYLINE or FILL_POLY.

The function call is **setup_poly(cmd)**, where *cmd* is the command (N_POLYLINE or N_FILL_POLY) to be processed.

setup_poly does the following:

- Calls get_thing to retrieve the data in the message.
- Validates the parameters.
- Determines if the component descriptor table has room available.
- Places the polygon's line and point data into the component descriptor table.

If successful, the function returns *rtn_val* set to 0 (SUCCESS). If an error is detected, the *rtn_val* is one of the following:

- 88 (COMPONENT_DESCRIPTOR_TBL_FULL)
The table does not have enough room for the new data.
- 99 (SYNTAX_ERROR)
The data in the message could not be processed.

Called By: process_command

Routines Called: get_thing

printf

Parameters:

int

cmd

Returns:

rtn_val

2.2.12 proc_cmd.c (process_command)

The process_command function retrieves a command string using get_thing, then calls the appropriate setup_* routine to process it.

The function call is **process_command()**. process_command does the following:

- Calls get_thing to retrieve a command string.
- Compares the string with each possible command to determine which it is.
- When a match is found, calls the applicable setup routine.
- Repeats the loop until all commands in the input file have been retrieved.

The commands processed by process_command, and the setup function it calls for each, are listed below.

Command	Function Called(cmd)
A_BIT_BLT or B_BIT_BLT	setup_bit_blt(N_BIT_BLT)
A_COMP_START or B_COMP_START	setup_comp_start(N_COMP_START)
A_DEFINE_STRING or B_DEFINE_STRING	setup_define_string(N_DEFINE_STRING)
A_DEFINE_WINDOW or B_DEFINE_WINDOW	setup_define_window(N_DEFINE_WINDOW)
A_DRAW_LINE or B_DRAW_LINE	setup_draw_line(N_DRAW_LINE)
A_DRAW_OVAL or B_DRAW_OVAL	setup_oval_rectangle(N_DRAW_OVAL)
A_DRAW_RECT or B_DRAW_RECT	setup_oval_rectangle(N_DRAW_RECT)
A_ENDCOMP or B_ENDCOMP	none
A_FILL_OVAL or B_FILL_OVAL	setup_oval_rectangle(N_FILL_OVAL)
A_FILL_POLY or B_FILL_POLY	setup_poly(N_FILL_POLY)
A_FILL_RECT or B_FILL_RECT	setup_oval_rectangle(N_FILL_RECT)
A_POLYLINE or B_POLYLINE	setup_poly(N_POLYLINE)
A_TEXT or B_TEXT	setup_text(N_TEXT)

The *rtn_val* returned by process_command is the value returned from the last setup_* function called. If no errors are detected, therefore, the *rtn_val* is 0 (SUCCESS).

If an error is detected by any called procedure, process_command increments an error counter. If the error count exceeds MAX_COMPILE_ERRORS (defined in defines_2d.h), process_command returns a *rtn_val* of 96 (TOO_MANY_ERRORS). This causes compile_2d to terminate the compile.

Called By: compile_2d

Routines Called: get_thing
 printf
 setup_bit_blt
 setup_comp_start
 setup_define_string
 setup_define_window
 setup_draw_line
 setup_oval_rectangle
 setup_poly
 setup_text
 strcmp

Parameters: none

Returns: rtn_val

2.2.13 **string.c (setup_define_string)**

The `setup_define_string` function places initial string data into the component descriptor table. This function is called by `process_command` if the 2-D command to be processed is `DEFINE_STRING`.

The function call is `setup_define_string(cmd)`, where *cmd* is the command (`N_DEFINE_STRING`) to be processed.

`setup_define_string` does the following:

- Verifies that component start data has been entered into the component descriptor table.
- Calls `get_thing` to retrieve the data in the message.
- Determines whether the component descriptor table has room available.
- Places the string's font, x coordinate, and y coordinate into the component descriptor table.
- If the string's length is odd, adds 1 to make it even.
- Makes sure the string is shorter than the allowed maximum.
- Puts the string into the component descriptor table.

If successful, the function returns *rtn_val* set to 0 (SUCCESS). If an error is detected, the *rtn_val* is one of the following:

88 (COMPONENT_DESCRIPTOR_TBL_FULL)

The table does not have enough room for the new data.

99 (SYNTAX_ERROR)

The string exceeds the maximum length allowed, the string contains a non-ASCII character, or the data in the message cannot be processed.

Called By:	process_command	
Routines Called:	get_thing printf strlen	
Parameters:	int	cmd
Returns:	rtn_val	

2.2.14 text.c (setup_text)

The `setup_text` function is responsible for placing fixed-length text data into the component descriptor table. This function is called by `process_command` if the 2-D command to be processed is TEXT.

The function call is `setup_text(cmd)`, where `cmd` is the command (N_TEXT) to be processed.

`setup_text` does the following:

- Calls `get_thing` to retrieve the data in the message.
- Verifies that the component descriptor table has room available.
- Places the text's font, x coordinate (lower left column), and y coordinate (lower left row) into the component descriptor table.
- If the text string's length is odd, adds 1 to make it even.
- Places the text string into the component descriptor table.

If successful, the function returns `rtn_val` set to 0 (SUCCESS). If an error is detected, the `rtn_val` is one of the following:

88 (COMPONENT_DESCRIPTOR_TBL_FULL)

The table does not have enough room for the new data.

99 (SYNTAX_ERROR)

The text string contains a non-ASCII character, or the data in the message cannot be processed.

Called By:	process_command	
Routines Called:	get_thing printf strlen	
Parameters:	int	cmd
Returns:	rtn_val	

2.2.15 **u_comp_2d.c (compile_2d)**

The `compile_2d` function in the `u_comp_2d.c` CSU is the offline equivalent of the `compile_2d` function in the `cig_comp_2d.c` CSU. This version of `compile_2d` can be run offline to compile an ASCII 2-D database file that was created manually.

The function call is **`compile_2d()`**. `compile_2d` does the following:

- Calls `init_stuff` to initialize various compiler variables.
- Calls `get_msg_2d` to get the first line of the input file.
- Calls `process_command` to process each command from the input file.
(`process_command` calls `get_thing` which in turn calls `get_msg_2d` to get each message from the file.)
- Checks for errors from `process_command`.
- Calls `linkup` to set up the window pointers and create the downloadable binary file.
- Closes the input and output files.
- Reports the number of errors detected during the compile.

Called By: `main`

Routines Called: `close`
`fclose`
`get_msg_2d` (offline version)
`init_stuff`
`linkup` (offline version)
`printf`
`process_command`

Parameters: `none`

Returns: `none`

2.2.16 **u_getm_2d.c (get_msg_2d)**

The `get_msg_2d` function in the `u_getm_2d.c` CSU is the offline equivalent of the `get_msg_2d` function in the `cig_getm_2d.c` CSU. This function gets the next 2-D message from the ASCII input file created offline, and sets a pointer to it for the 2-D compiler.

The function call is **`get_msg_2d()`**. The `msg_code` returned is 97 (END_OF_FILE) if the input file contains no more lines, or 0 (SUCCESS) if a line is read successfully.

Called By: `compile_2d` (offline version)
`get_thing`

Routines Called: `fgets`

Parameters: none

Returns: msg_code

2.2.17 u_link_2d.c (linkup)

The linkup function in the u_link_2d.c CSU is the offline equivalent of the linkup function in the cig_link_2d.c CSU. This function sets up window pointers and allocates available MPV memory for windows. It also saves all of the 2-D data to a binary file that can be copied to the CIG and downloaded to the MPV board at a later time.

The function call is `linkup()`. linkup does the following:

- Verifies that the number of component start entries equals the number of component end entries.
- Calculates base addresses and table sizes for all information.
- Outputs the following data to stdout (see Figure 2-6 for a sample output):
 - Component pointers table base address and size.
 - Window descriptor table base address and size.
 - Component descriptor table base address and size.
 - Allocatable window base address and maximum size.
 - Base program address.
- Sets up the screen window area (this should not vary).
- Changes the component pointers to absolute addresses.
- Allocates space for the dynamic polygon buffer areas.
- Sets the allocatable window area to the space following the component descriptor table.
- Allocates space for all windows and sets the window pointers.
- Writes all data to the 2-D binary database file: headers, window structures, component pointer table, window descriptor table, and component descriptor table.

Called By: compile_2d (offline version)

Routines Called: printf
write

Parameters: none

Returns: none

2.2.18 u_main2d.c (main)

The main function is the driver for the 2-D compiler when it is run offline to create overlays from a manually built ASCII file.

main is invoked by the user. The user may specify the name of the ASCII database file (input) and the name of the binary database file (output) as arguments on the command line. main does the following:

- Prompts for the input and output file names if not entered on the command line.
- Opens the specified input file.
- Creates the specified output file.
- Calls compile_2d.

Called By: none (invoked by user)

Routines Called: compile_2d (offline version)
 creat
 fopen
 printf
 scanf
 strcpy

Parameters: int argc
 char *argv[]

Returns: none

2.2.19 window.c (setup_define_window)

The setup_define_window function places window data into the window descriptor table. This function is called by process_command if the 2-D command to be processed is DEFINE_WINDOW.

The function call is setup_define_window(cmd), where cmd is the command (N_DEFINE_WINDOW) to be processed.

setup_define_window does the following:

- Calls get_thing to retrieve the data in the message.
- Verifies that the parameters are valid.
- Computes the window's pitch and conversion factor.
- Places all window parameters (number of horizontal pixels, number of vertical pixels, pitch, and GSP conversion factor) into the window array structure.
- Places the window number into the allocation list so linkup can allocate memory for the window.

Pitch and conversion factors are computed as shown below. "dx" is the number of horizontal pixels.

dx range (hex)	pitch (hex)	count (dec)	conversion factor (dec)
4001-8000	8000	15	16
2001-4000	4000	14	17
1001-2000	2000	13	18
801-1000	1000	12	19
401-800	800	11	20
201-400	400	10	21
101-200	200	9	22
80-100	100	8	23
41-80	80	7	24
21-40	40	6	25
11-20	20	5	26
8-10	10	4	27
4-8	8	3	28
2-4	4	2	29
1-2	2	1	30
1-1	1	0	31

If successful, the function returns a *rtn_val* of 0 (SUCCESS). If an error occurs, the *rtn_val* is one of the following:

- 1 (INVALID_WINDOW_NUMBER) The window number is out of range.
- 2 (INVALID_WINDOW_DX) The window's width is out of range.
- 3 (INVALID_WINDOW_DY) The window's height is out of range.
- 4 (WINDOW_PITCH_TOO_LARGE) The window's pitch is out of range.
- 99 (SYNTAX_ERROR) The data in the message cannot be processed.

Called By: process_command

Routines Called: get_thing
printf

Parameters: int cmd

Returns: rtn_val

2.3 Backend Manager (/cig/libsrc/libbackend)

The Backend Manager CSC is responsible for configuring and managing the backend of a GT100 CIG. It gives the real-time software a hardware-independent interface to the boards in the backend.

The major components of the backend include:

- AAM (Active Area Memory)
- ESIFA (Enhanced Subsystem Interface Adapter)
- EVC (Ethernet VME Controller)
- MPV (Micro Processor Video)
- PPM (Pixel Processor Memory)

Commands from the real-time software to the subsystem boards are sent to the backend manager. Many of these commands are the result of messages received from the Simulation Host. The backend manager then calls the appropriate functions to interface with each board.

The processes managed by the backend manager include the following:

- Initializing active area memory.
- Loading new branch values into active area memory.
- Loading new system view flags into active area memory.
- Changing viewport modifiers (thermal white hot, thermal black hot, etc.)
- Loading new color lookup tables.
- Processing laser range requests.
- Turning video channels off and on.
- Downloading files to the PPM.

During a simulation, requests for the MPV and the ESIFA are retained in queues. At the end of each frame, the backend manager triggers the subsystem boards to process the messages in their respective queues.

Each backend in the CIG has its own backend manager. The backend objects are defined in the `be_table` array. Each element in `be_table[]` specifies the following:

- The backend id.
- A flag indicating whether the backend is a T or a TX.
- The starting address of the backend's active area memory.
- An array of `laser_request` flags, one for each channel. These flags indicate whether laser depth processing has been requested for the channel.

At the current time, one CIG can contain up to two backends.

Figure 2-7 identifies the CSUs in the Backend Manager CSC. The functions performed by these CSUs are described in this section.

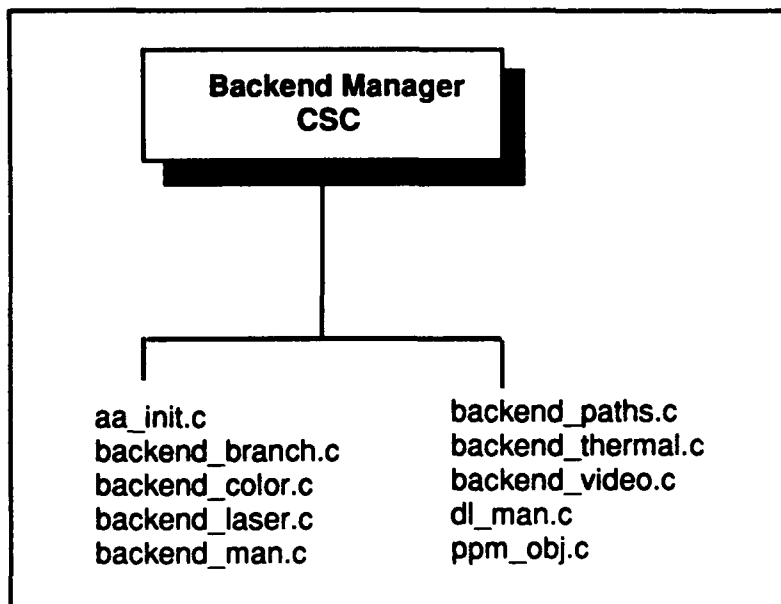


Figure 2-7. Backend Manager CSUs

2.3.1 `aa_init.c`

The functions in the `aa_init.c` CSU are used to initialize and clear active area memory. These functions are:

- `active_area_init`
- `clear`
- `extended_ram_available`

2.3.1.1 `active_area_init`

The `active_area_init` function initializes active area memory. Each backend has its own AAM. This function is called at start-up and when a simulation is ended.

The function call is `active_area_init(aam_addr)`, where *aam_addr* is the starting address of active area memory.

`active_area_init` does the following:

- Calls `bus_error` to verify that the specified active area memory address exists.
- Calls `clear` to zero out the system area of memory.
- If this is AAM1 (backend 0), calls `dl_setup` to initialize the dynamic state tables and the multiple-effects list.
- If this is not AAM1, calls `blcopy` to block copy AAM1 to the specified AAM.
- Calls `clear` to zero out the extended load module region.

The function returns 0 if successful. It returns EOF if the specified address does not exist.

Called By:	backend_reset backend_setup	
Routines Called:	blcopy bus_error clear dl_setup	
Parameters:	AAM	*aam_addr
Returns:	0 EOF	

2.3.1.2 clear

The clear function clears (zeroes out) a specified area and amount of memory. This function is used to clear the system portion of active area memory and the extended load module region at initialization time.

The function call is **clear(ptr, size)**, where:

ptr is a pointer to the beginning of the area to be cleared
size is the amount of memory (in bytes) to be cleared

Called By:	active_area_init	
Routines Called:	none	
Parameters:	UNS_4 UNS_4	*ptr size
Returns:	none	

2.3.1.3 extended_ram_available

The **extended_ram_available** function determines how many megabytes of additional RAM exist, starting at a specified location. This function is used to determine whether the terrain database will fit into available memory.

The function call is **extended_ram_available(start_addr)**, where *start_addr* is the starting memory address.

extended_ram_available adds 1 megabyte to the starting address and calls **bus_error** to see if that address exists. It repeats this process until it has checked 6 megabytes or an address

has been found to be non-existent. It then returns the amount of additional memory found (the last valid address checked minus the starting address).

Called By: open_dbase

Routines Called: bus_error

Parameters: UNS_1 *start_addr

Returns: i - start_addr

2.3.2 backend_branch.c (backend_set_branch)

The backend_set_branch function loads new branch values into active area memory. This function is called when the Simulation Host sends a MSG_VIEW_FLAGS message to change the branch values used to select conditional traversal paths in the configuration tree.

The function call is **backend_set_branch(branchp, size, db)**, where:

branchp is a pointer to the new view flags/branch values array

size is the size of the view flags/branch values array

db is the current double-buffer base pointer

backend_set_branch sets a pointer in active area memory, then copies the new view flags/branch values into active area memory.

Called By: msg_view_flags

Routines Called: bcopy

Parameters: UNS_4 *branchp
UNS_2 size
UNS_4 db

Returns: none

2.3.3 backend_color.c (backend_set_color)

The backend_set_color function is responsible for controlling the color mapping on a specified backend. The color map is defined via the downloaded color configuration file. This function is called when the Simulation Host sends a MSG_SUBSYS_MODE message to change lookup tables.

The function call is **backend_set_color** (**backend**, **channel**, **lut3d**, **lut2d**, **sky_and**, **sky_or**, **laser_and**, **laser_or**, **dtp_th_word**), where:

backend is the subsystem to be affected

channel is the channel to be affected

lut3d is the new three-dimensional color lookup table (for TX backends only)

lut2d is the new two-dimensional color lookup table (for TX backends only)

sky_and is the and_sky fade value (for T backends only)

sky_or is the or_sky fade value (for T backends only)

laser_and is the and_laser fade value (for T backends only)

laser_or is or_laser fade value (for T backends only)

dtp_th_word is the DTP thermal switching word

backend_set_color does the following:

- Calls **backend_get_object_addr** to get a pointer to the specified backend.
- For a TX backend, calls **mpvideo_set_lut** to set the **lut3d** (3-D lookup table) and **lut2d** (2-D lookup table).
- Calls **esifa_set_special** to set the bits for the ESIFA layouts (**sky_and**, **sky_or**, **laser_and**, **laser_or**).
- Sets the DTP thermal switching word for the entire subsystem based on the value in **COLOR.CFG**.

The function returns 0 if successful. It returns EOF if the address returned for the specified backend is NULL.

Called By: **msg_subsys_mode**

Routines Called: **backend_get_object_addr**
esifa_set_special
mpvideo_set_lut
printf

Parameters:	UNS_4	backend
	UNS_2	channel
	UNS_2	lut3d
	UNS_2	lut2d
	UNS_2	sky_and
	UNS_2	sky_or
	UNS_2	laser_and
	UNS_2	laser_or
	UNS_4	dtp_th_word

Returns: **0**
EOF

2.3.4 backend_laser.c

The functions in **backend_laser.c** handle laser range requests. These functions are:

- `backend_laser_request_range`
- `backend_response`

For TX backends, the Simulation Host can specify a pixel location. Laser range requests are sent to the MPV via the MPV Interface routines and the Force board.

For T backends, a hard-wired pixel location is used. Laser range requests are sent to and processed by the ESIFA.

2.3.4.1 `backend_laser_request_range`

The `backend_laser_request_range` function is used to enable laser range calculations (pixel depth data for a specific screen position). If laser range processing is enabled, each frame the CIG returns the distance from the viewpoint to the object within the requested pixel. For T backends, which cannot select pixel positions, a hard-wired pixel is used to determine range. This function is called when the Simulation Host sends a `MSG_LASER_REQUEST_RANGE` message.

The function call is `backend_laser_request_range(backend, channel, i, j, id)`, where:

backend is the subsystem id
channel is the channel number within the specified subsystem
i is the horizontal position of the pixel for which the range is to be returned
j is the vertical position of the pixel for which the range is to be returned
id is an identifier to be attached to the laser return message

`backend_laser_request_range` does the following:

- Calls `backend_get_object_addr` to get a pointer to the specified backend.
- For a TX backend:
 - Calls `mpvideo_laser_request_range` to set the pixel position and id.
 - Sets the `laser_request` flag for the specified channel to TRUE.
- For a T backend:
 - Sets the `laser_request` flag for every channel in the backend to FALSE.
 - Calls `esifa_laser_request_range` to set up the ESIFA for laser range processing.
 - Sets the `laser_request` flag for the specified channel to TRUE.

The function returns 0 if successful. It returns EOF if the address returned for the specified backend is NULL, or if the channel cannot be found.

Called By: `msg_laser_request_range`

Routines Called: `backend_get_object_addr`
`esifa_laser_request_range`
`mpvideo_laser_request_range`
`printf`

Parameters:	UNS_4	backend
	UNS_4	channel
	INT_2	i
	INT_2	j
	INT_2	id

Returns:	0
	EOF

2.3.4.2 backend_response

The `backend_response` function returns a previously requested laser range from a specified backend channel. This function is called at the end of every frame.

The function call is `backend_response(backend, lm_blk)`, where:

backend is the backend for which laser range information was requested
lm_blk is the number of load modules per side of a load module block; this value is used to scale the range if load module blocking is enabled

`backend_response` does the following:

- Calls `backend_get_object_addr` to get a pointer to the specified backend.
- For a TX backend:
 - Calls `mpvideo_response` to get the laser range data; `mpvideo_response` then calls `msg_laser_return` to generate the return message for the Simulation Host.
- For each channel on a T backend:
 - If the `laser_request` flag is enabled for the channel, calls `esifa_laser_return` to get the laser range value.
 - Calls `msg_laser_return` to generate the message to be returned to the Simulation Host.

The function returns 0 if successful. It returns EOF if the address returned for the specified backend is NULL.

Called By:	<code>msg_end</code>
------------	----------------------

Routines Called:	<code>backend_get_object_addr</code>
	<code>esifa_laser_return</code>
	<code>mpvideo_response</code>
	<code>msg_laser_return</code>

Parameters:	UNS_4	backend
	INT_4	lm_blk

Returns:	0
	EOF

2.3.5 backend_man.c

The functions in the backend_man.c CSU form the backend manager library. These functions are the following:

- backend_setup
- backend_reset
- backend_sim_init
- backend_send_req
- backend_get_object_addr
- backend_clear_laser_requests

2.3.5.1 backend_setup

The backend_setup function allocates memory for an instance of the backend object, then calls various functions to configure the backend.

The function call is **backend_setup(backend, aam_addr, mpvio_boot, mpvio_mbx, esifa)**, where:

backend is the id of the backend to be initialized
aam_addr is the starting VMEbus address of the backend's active area memory
mpvio_boot is the base VMEbus address of the MPV I/O board (TX backends only)
mpvio_mbx is the VMEbus address of the MPV I/O interface mailbox (TX backends only)
esifa is the ESIFA device name

backend_setup does the following:

- Calls active_area_init to initialize active area memory.
- Allocates memory for the new backend object and sets a pointer to it.
- Calls backend_clear_laser_requests to clear any laser requests for the backend.
- Calls esifa_setup to establish a communications path with the ESIFA board.
- Calls ppm_setup to initialize the PPM object for each PPM in the backend.
- Calls mpvideo_setup to initialize the MPV object; if an error is returned, sets the backend type to T.
- Calls load_esifa to download the textures.lst file to the ESIFA board.

The function returns the status returned from esifa_setup OR'd with the status returned from mpvideo_setup. It returns EOF if the *aam_addr* does not exist.

Called By: initialize

Routines Called: active_area_init
 backend_clear_laser_requests
 esifa_setup
 free
 load_esifa
 malloc

mpvideo_setup
ppm_setup
printf

Parameters:	UNS_4 AAM UNS_4 UNS_4 UNS_1	backend *aam_addr mpvio_boot mpvio_mbx *esifa
-------------	---	---

Returns: EOF
(esifa_error | mpv_error)

2.3.5.2 backend_reset

The backend_reset function resets the backend(s) to their initial states. This function is called when the Simulation Host sends a CIG Control-Stop message.

The function call is **backend_reset()**. For each backend in the CIG, backend_reset does the following:

- For a TX backend:
 - Calls mpvideo_stop to halt the MPV's 2-D processor task.
 - Calls mpvideo_set_video to turn the video off.
- For a T backend:
 - Calls esifa_set_video to turn each channel's video off.
 - Calls esifa_send_req to perform any requested I/O with the ESIFA board.
- Calls backend_clear_laser_requests to clear any pending laser range requests for the backend.
- Calls active_area_init to clear the backend's active area memory.

Called By: msg_cig_ctl

Routines Called: active_area_init
backend_clear_laser_requests
esifa_send_req
esifa_set_video
mpvideo_set_video
mpvideo_stop

Parameters: none

Returns: none

2.3.5.3 backend_sim_init

The `backend_sim_init` function prepares the backend to run a simulation. This function is called before beginning a simulation. It notifies other subsystem objects that the system is about to enter the simulation state.

The function call is `backend_sim_init()`. `backend_sim_init` does the following:

- Calls `mpvideo_sim_init` to prepare the MPV for a simulation.
- Calls `esifa_sim_init` to prepare the ESIFA for a simulation.

Called By: `init_simulation`

Routines Called: `esifa_sim_init`
`mpvideo_sim_init`

Parameters: `none`

Returns: `none`

2.3.5.4 backend_send_req

The `backend_send_req` function sends queued messages to the backend. This function is called at the end of each frame. It initiates the processing of all messages that accumulated during the previous frame.

The function call is `backend_send_req()`. `backend_send_req` does the following:

- Calls `backend_update_view_paths` to update the system view flags in active area memory.
- Calls `mpvideo_send_req` to trigger processing of all queued MPV requests.
- Calls `esifa_send_req` to trigger processing of all queued ESIFA requests.

Called By: `msg_end`

Routines Called: `backend_update_view_paths`
`esifa_send_req`
`mpvideo_send_req`

Parameters: `none`

Returns: `none`

2.3.5.5 backend_get_object_addr

The `backend_get_object_addr` function returns a pointer to a specified backend object. The function returns `NULL` if the backend is not in the `be_table` array.

The function call is `backend_get_object_addr(backend)`, where *backend* is the backend id.

Called By:

- backend_laser_request_range
- backend_response
- backend_set_color
- backend_set_thermal
- backend_set_video
- backend_update_view_paths
- be_query_num_paths
- cal

Routines Called: none

Parameters: UNS_4 backend

Returns: be_table[n]
NULL

2.3.5.6 backend_clear_laser_requests

The `backend_clear_laser_requests` function clears any pending laser range requests for a backend by setting the `laser_request` flag for every channel to `FALSE`. This function is called whenever the backend is initialized or reset.

The function call is **backend_clear_laser_requests(pbe)**, where *pbe* is a pointer to the backend object.

Called By: backend_reset
backend_setup

Routines Called: none

Parameters: BACKEND_OBJ *pbe

Returns: `none`

2.3.6 backend_paths.c

The functions in backend_paths.c are used to load changes to the system view flags into active area memory. These functions are:

- backend_set_paths
- backend_update_view_paths

2.3.6.1 backend_set_paths

The backend_set_paths function copies new system view flags to a temporary location. This function is called when the Simulation Host sends a MSG_VIEW_FLAGS message to update the system view flags (on/off, EO, FLIR, DTV, etc.). At the end of the frame, backend_update_view_paths uses the data set by backend_set_paths to update the system view flags in active area memory.

The function call is **backend_set_paths(flags)**, where *flags* is the new system view flags array.

Called By:	msg_view_flags	
Routines Called:	none	
Parameters:	UNS_4	flags
Returns:	none	

2.3.6.2 backend_update_view_paths

The backend_update_view_paths function copies the new system view flags from the temporary location used by backend_set_paths into active area memory. This function is called at the end of each frame.

The function call is **backend_update_view_paths()**.

Called By:	backend_send_req
Routines Called:	AAM2_ADDR backend_get_object_addr
Parameters:	none

Returns: none

2.3.7 backend_thermal.c (backend_set_thermal)

The backend_set_thermal function handles backend thermal changes. This function is called when the Simulation Host sends a MSG_VIEWPORT_UPDATE message to change viewport modifier information (thermal white hot, thermal black hot, etc.).

The function call is **backend_set_thermal(backend, channel, thermal_flag, white_hot_flag, dtp)**, where:

backend is the id of the backend of the viewport to be changed

channel is the id of the channel to be changed

thermal_flag is 0 (alternate mode is disabled) or 1 (alternate mode is enabled)

white_hot_flag is 0 (modifier to alternate is off) or 1 (modifier to alternate is on)

dtp is the DTP mode (alternate, modifier, or normal)

backend_set_thermal does the following:

- Calls backend_get_object_addr to get a pointer to the specified backend.
- For a TX backend:
 - Calls mpvideo_num_paths to determine the number of graphics paths.
- For a T backend:
 - Sets the number of graphics paths to 1.
- For each path:
 - Calls esifa_set_thermal to put the changes into effect.
 - Sets the DTP channel thermal value.

The function returns 0 if successful. It returns EOF if the backend object cannot be found.

Called By: msg_viewport_update

Routines Called: AAM2_ADDR
 backend_get_object_addr
 esifa_set_thermal
 mpvideo_num_paths
 printf

Parameters:	UNS_4	backend
	UNS_4	channel
	UNS_4	thermal_flag
	UNS_4	white_hot_flag
	UNS_4	dtp

Returns: EOF

2.3.8 backend_video.c (backend_set_video)

The `backend_set_video` function turns a specified channel on or off. This function is called when the Simulation Host sends a `MSG_VIEWPORT_UPDATE` message to change viewport modifier information (thermal white hot, thermal black hot, etc.).

The function call is `backend_set_video(backend, channel, flag)`, where:

backend is the backend/subsystem id (0 or 1)

channel is the channel id (for subsystem 1, this is the viewport id; for subsystem 0, this is the viewport id minus 8)

flag is 0 (off) or 1 (on)

`backend_set_video` does the following:

- Calls `backend_get_object_addr` to get a pointer to the specified backend.
- For a TX backend:
 - Calls `mpvideo_set_video` to turn the specified channel on or off.
 - Calls `esifa_set_video` to turn the specified channel on or off.
- For a T backend:
 - Calls `esifa_set_video` to turn the specified channel on or off.

The function returns 0 if successful. It returns EOF if the backend cannot be found.

Called By: `msg_viewport_update`

Routines Called: `backend_get_object_addr`
`esifa_set_video`
`mpvideo_set_video`
`printf`

Parameters:	<code>UNS_4</code>	<code>backend</code>
	<code>UNS_4</code>	<code>channel</code>
	<code>UNS_4</code>	<code>flag</code>

Returns: 0
 EOF

2.3.9 dl_man.c (dl_setup)

The `dl_setup` function is responsible for initializing the static and dynamic state tables and the multiple-frame effects linked list. This function is called during CIG configuration.

The function call is `dl_setup()`. `dl_setup` does the following:

- Initializes tanks and other vehicles in the dynamic state table.
- Initializes tanks and other vehicles in the static state table.

- Initializes the multiple-frame effects linked list. (This structure is used when showing effects over multiple frames.)

Called By: active_area_init

Routines Called: INIT_MTX

Parameters: none

Returns: none

2.3.10 ppm_obj.c

The functions in ppm_obj.c are used to download files to the PPM (Pixel Processor Memory) board. These functions are:

- ppm_setup
- ppm_init
- ppm_get_data
- ppm_load
- gos_ppm_query
- gos_ppm_query_menu

The data for each PPM object is maintained in the ppm_data[] array. This array is indexed by backend id and PPM index. Each element in the array specifies the PPM's configuration file name, display modes (A and B), display offsets (A and B), pixel location (i, j), board address, screen position, calibration pixel address, and screen size.

2.3.10.1 ppm_setup

The ppm_setup function initializes the ppm_data array and creates a PPM object for each PPM in each backend. This function is called when the backend is initialized. It is also called by the other ppm_obj functions if they detect that a PPM object has not yet been initialized.

The function call is **ppm_setup()**. For each PPM in each backend, the function does the following:

- Initializes the file name to "UNKNOWN".
- Initializes the A and B display modes to 0.
- Initializes the A and B display offsets (i, j) to 16,48.
- Initializes the pixel location (i, j) to 100,100.
- Initializes the board address based on the PPM index.
- Initializes the screen position.
- Initializes the calibration pixel address.
- Initializes the screen size.
- Sets ppm_obj_initialized to TRUE.

Called By: backend_setup
 gos_ppm_query
 ppm_get_data
 ppm_init
 ppm_load

Routines Called: none

Parameters: none

Returns: none

2.3.10.2 ppm_init

The ppm_init function updates the PPM object's structure with the information it receives from esifa_load. This function is called when the ESIFA configuration file is downloaded to the ESIFA.

The function call is **ppm_init(backend, ppm_bit_mask, ppm_fn, board_address, data_P)**, where:

backend is the backend id

ppm_bit_mask is **BRD0** (PPM index 0), **BRD1** (PPM index 1), **BRD2** (PPM index 2), or **BRD3** (PPM index 3)

ppm_fn is the name of the PPM configuration file

board_address is the address of the PPM

data_P is a pointer to additional data to be loaded into the PPM object; this data is either display mode, screen position, screen size, pixel address, or pixel state, based on the PPM mode

The function does the following:

- If the PPM object has not yet been initialized, calls ppm_setup.
- Determines the PPM's index based on the specified ppm_bit_mask.
- Sets the PPM object's file name to the name specified.
- Sets the PPM object's board address to the address specified.
- Determines the PPM mode based on the board address and the PPM_MODE_MASK (defined in ppm.h).
- Uses the data passed in *data_P to set the appropriate variable(s) in the PPM object, based on the PPM mode.

Called By: esifa_load

Routines Called: ppm_setup
 printf

Parameters:	UNS_4	backend
	INT_4	ppm_bit_mask
	char	*ppm_fn
	UNS_4	board_address
	UNS_2	*data_P

Returns: none

2.3.10.3 ppm_get_data

The `ppm_get_data` function returns a pointer to a specified PPM object. This function is called by the Message Processing functions that handle PPM messages from the Simulation Host. It is also called if the Gossip user selects the i ("PPM info") option from the PPM Query menu.

The function call is `ppm_get_data(backend, ppm_index)`, where:

backend is the backend id

ppm_index is the PPM index (within the specified backend)

`ppm_get_data` calls `ppm_setup` if it detects that the PPM object has not been initialized. It then returns a pointer to the PPM object's structure.

Called By:	gos_ppm_query
	msg_ppm_pixel_state
	msg_ppm_display_mode
	msg_ppm_display_offset
	msg_ppm_pixel_location

Routines Called: `ppm_setup`

Parameters:	UNS_4	backend
	UNS_4	ppm_index

Returns: `&ppm_data[backend][ppm_index]`

2.3.10.4 ppm_load

The `ppm_load` function is used to download data to the PPM via the ESIFA board. This function is called if the Gossip user selects the p ("ppm load") option from the PPM Query menu.

The function call is `ppm_load(backend, ppm_address, ppm_data)`, where:

backend is the backend id

ppm_address is the PPM board's address

ppm_data is the data to be loaded into the PPM

ppm_load does the following:

- If the PPM object has not been initialized, calls ppm_setup.
- Gets a pointer to the ESIFA object for the specified backend.
- Calls esifa_ConfigData to get a pointer to the ESIFA's configuration data.
- Calls esifa_write to load the specified data into ESIFA RAM.
- Calls esifa_download to download the data from ESIFA RAM to the PPM board.

The function returns EOF if the ESIFA object cannot be found or is not initialized.

Called By: gos_ppm_query

Routines Called: esifa_ConfigData
esifa_download
esifa_get_object_addr
esifa_write
ppm_setup
printf

Parameters:	UNS_4	backend
	UNS_4	ppm_address
	UNS_2	ppm_data

Returns: EOF

2.3.10.5 gos_ppm_query

The gos_ppm_query function provides a user interface to display and modify PPM data. This function is called if the Gossip user selects the P ("PPM query") option from the Gossip main menu.

The function call is gos_ppm_query(). gos_ppm_query does the following:

- If the PPM object is not initialized, calls ppm_setup.
- Calls gos_ppm_query_menu to display the PPM Query menu.
- Uses unbf_getchar to get the keystroke entered by the user.
- Processes the user's request (see table below).
- Displays the "ppm_query>" prompt.

The following table lists the options supported by gos_ppm_query, and shows the steps it performs in response to each user selection.

PPM Query Menu Option	Processing by gos_ppm_query
? print this menu	Calls gos_ppm_query_menu.
B blank display	Clears screen.
b change backend	Prompts user for new backend id; sets internal variable.
c change ppm	Prompts user for new ppm index; sets internal variable.
d Download parameters	Prompts user for ESIFA RAM address, byte count, and ppm address; calls esifa_download.
e esifa info	Calls esifa_ConfigData; displays data returned.
i PPM info	Calls ppm_get_data; displays data returned.
p ppm load	Prompts user for ppm address and ppm data; calls ppm_load.
r read ESIFA ram	Prompts user for ESIFA address and number of bytes; calls esifa_read; displays data returned.
w write ESIFA ram	Prompts user for ESIFA address and number of bytes; calls esifa_write.
x exit/quit	Exits.

Called By: gossip_tick

Routines Called: blank
 cup
 esifa_ConfigData
 esifa_download
 esifa_read
 esifa_write
 gos_ppm_query_menu
 ppm_get_data
 ppm_load
 ppm_setup
 printf
 scanf
 unbf_getchar

Parameters: none

Returns: none

2.3.10.6 gos_ppm_query_menu

The gos_ppm_query_menu function displays the PPM Query menu. This function is called by gos_ppm_query when it is first invoked or if the user later enters ? (help) on the command line.

The function call is gos_ppm_query_menu(). The function does the following:

- Clears the screen.
- Displays the PPM Query menu options.
- Displays the "ppm_query>" prompt.

For a list of the options displayed on the menu, see gos_ppm_query.

Called By: gos_ppm_query

Routines Called: blank
cup
printf

Parameters: none

Returns: none

2.4 Ballistics Processing (/cig/libsrc/libball)

The Ballistics Processing CSC is the part of the GT Host Software that is responsible for the following:

- Detecting intersections with the terrain database and the currently viewable models (static and dynamic vehicles).
- Processing round data and returning hit or miss information to the real-time software.
- Processing trajectory chord data and returning hit or miss information to the real-time software.
- Providing terrain feedback data for points on specified vehicles in active area memory. The data returned identifies the database polygon or model located at the vehicle's current position.

In order to compute intersections with the terrain and simulation models, the Ballistics Processing CSC acquires and maintains polygon and bounding volume information from the terrain database, as well as static vehicle information from the real-time software.

Ballistics Processing may be run on a master board or a slave board in the CIG, as follows:

Master

If the CIG has only one MVME147 board, it is the master that is used to run all of the real-time software, including Ballistics.

Slave

If the CIG has two MVME147 boards, one board is the master that runs the real-time software. The other board is the slave that runs Ballistics. This configuration is used for high rate-of-fire weapons.

The Ballistics software that runs on a Master board is very similar to the software that runs on a Slave board. The differences are identified in the source code by compiler flags. The real-time software determines what type of Ballistics board is in the CIG, then loads the appropriate version of the Ballistics task.

The major data structures used in Ballistics Processing are the following:

Trajectory table directory

Contains one entry for each trajectory table. A trajectory table, which describes the trajectory for a specific type of round, consists of the trajectory type, frame rate, effect type, table size, and a pointer to the table's entries. Each trajectory table entry contains the trajectory's boresight x and y coordinates (with respect to the gun barrel).

Trajectory tables are predefined for certain round types. The Simulation Host may define trajectory tables for other round types.

Terrain model directory

Describes the models that are placed on the terrain (houses, telephone poles, water towers, etc.). Each entry defines the model type, bvol flag, component count, bvol count, model directory type, model radius, and the primary, secondary, and tertiary bvol indices.

Note: The terrain model directory is not currently used. It is defined to accommodate future enhancements to the database.

Terrain bvol directory

Describes the bounding volume for each terrain model. Each entry defines the model directory type, type id, the bvol's height above the poly-defining perimeter, and the perimeter defining the bvol polygon (its vertices).

Note: The terrain bvol directory is not currently used. It is defined to accommodate future enhancements to the database.

DED model directory

Describes the models in the dynamic elements database. Each entry defines the model type, bvol flag, component count, bvol count, model directory type, model radius, and the primary, secondary, and tertiary bvol indices.

DED bvol directory

Describes the bounding volume for each DED model. Each entry defines the bvol index, the model directory type, type id, the bvol's height above the poly-defining perimeter, and the perimeter defining the bvol polygon (its vertices).

Load module directory

Contains one entry for each load module in active area memory. Each load module entry contains the load module's cache flag, frame stamp, polygon count, maximum polygon height above the poly-defining perimeter, bvol count, and maximum bvol height above the poly-defining perimeter. Each load module entry also contains pointers to the polygon and bvol lists attached to that load module.

Static vehicle directory

Contains one entry for every load module in active area memory. Each entry points to a list of the static vehicles in that load module. Each entry in the static vehicle list contains the static vehicle's vehicle id, active area memory partition index, component count, unique type, load module number, application-specific data (ASID), transformation matrix, rotation angles for the second component, and back and forward pointers.

Static vehicle entries that are not currently assigned to a load module are contained in the static vehicle free list. When the Simulation Host adds a static vehicle, Ballistics removes one from the free list and adds it to the proper load module list. When the Simulation Host deletes a static vehicle, Ballistics removes it from the load module and returns it to the free list. The free list is a mechanism for ensuring that the maximum number of static vehicles is not exceeded.

Polygon lists

Contain one entry for each polygon in a given load module in active area memory. Each entry contains the polygon's soil type, vertex count, priority, shade, minimum and maximum values, Ballistics flag, local terrain flag, grid location, and vertex list. Each load module in active area memory has its own polygon list.

Polygon entries that are not currently assigned to a load module are contained in the free polygon list. When a new load module is added to active area memory, Ballistics removes the required number of polygons from the free list and adds them to the new load module's polygon list. If the free list does not contain enough polygons for a new load module, Ballistics swaps out the least-recently-used load module. When a load module is removed from active area memory, Ballistics returns its polygons to the free list.

Bvol lists

Contain one entry for each bounding volume in a given load module in active area memory. Each entry contains the bvol's type id, distance above the poly-defining perimeter, vertex list, and grid location. Each load module in active area memory has its own bvol list.

bvol entries that are not currently assigned to a load module are contained in the free bvol list. When a new load module is added to active area memory, Ballistics removes the required number of bvols from the free list and adds them to the new load module's bvol list. If the free list does not contain enough bvols for a new load module, Ballistics swaps out the least-recently-used load module. When a load module is removed from active area memory, Ballistics returns its bvols to the free list.

Round list

Contains one entry for each active round. Each entry contains the round's active frame count, frame count, frame interval, trajectory entry index, trajectory table size, offset, trajectory pointer, points, and back and forward pointers.

Round entries that are not currently active are contained in the free round list. When the Simulation Host requests a new round, Ballistics removes one from the free list and adds it to the active list. After processing the round, Ballistics removes it from the active list and returns it to the free list. The free list is a mechanism for ensuring that the maximum number of rounds is not exceeded.

Terrain feedback point lists

Contain one entry for each point in a vehicle for which terrain feedback data is to be collected by Ballistics. Each entry contains the point number, current position, polygon data, and model data. Each vehicle for which the Simulation Host has requested terrain feedback processing has its own terrain feedback point list.

Feedback points that are not currently assigned to a vehicle are contained in the terrain feedback points free list. When the Simulation Host requests feedback processing for a vehicle, Ballistics removes the specified number of points from the free list and adds them to the vehicle's points list. When the Simulation Host disables feedback processing for a specified vehicle, Ballistics returns the vehicle's points to the free list. The free list is a mechanism for ensuring that the maximum number of feedback points is not exceeded.

The Ballistics task is created and started by the Task Initialization CSC. The Ballistics configuration file is processed by the config_ballistics function in the Real-Time Processing CSC, and Ballistics is put into the run state (when a simulation is started) by the sim_bal_start function.

The real-time software and Ballistics communicate by passing messages via the Ballistics message processing queues. This communication consists primarily of the following:

Messages sent from the Simulation Host

A typical message may tell Ballistics that a round has been fired or that a static vehicle has been added to the local terrain. Each Ballistics message is received by `process_a_msg` or `db_mcc_setup`, which pushes the message onto the incoming Ballistics message queue. For some messages, `process_a_msg` calls a specialized routine that performs some processing on the message, then pushes it onto the message queue. The routines that handle the messages going to Ballistics are contained in the `bal_routines.c` CSU; all are prefixed with the name `sim_bal`.

Ballistics processes the message (which typically involves computing whether any model or terrain in the database was hit), then returns a hit or miss message if applicable. Messages returned from Ballistics are removed from the outgoing message queue by the real-time software, which sends them to the Simulation Host.

New frame messages

Once per frame, the real-time software notifies Ballistics that a frame interrupt has taken place, and informs it (via a `MSG_B0_NEW_FRAME` message) of the current frame count and the new status of all dynamic vehicles.

Active area memory messages

At startup and whenever active area memory is moved, the real-time software notifies Ballistics (via a `MSG_B0_AAM_SW_CORNER` message) of the location of active area memory. Additionally, whenever the real-time software loads a new load module from disk into active area memory, it informs Ballistics using a `MSG_B0_LM_READ` message.

For the syntax and description of each Ballistics message passed between the Simulation Host and the real-time software, refer to the "GT100 CIG to Simulation Host Interface Manual."

The following table identifies all messages, in alphabetical order, that may be passed from the real-time software to Ballistics. The table also identifies the function(s) in the real-time software that push the message onto the Ballistics message queue, and the function in the Ballistics Message Processing component that processes that message.

The Ballistics functions referenced here are described later in this section. The real-time software functions are described in other sections of this document.

RTSW --> Ballistics Message	Sent By (RTSW Function)	Processed By (Ballistics Function)
MSG_B0_AAM_SW_CORNER	sim_bal_start, rowcol_rd	b0_aam_sw_corner
MSG_B0_ADD_STATIC_VEHICLE	sim_bal_static_add	b0_add_static_vehicle
MSG_B0_ADD_TRAJ_TABLE	db_mcc_setup, process_a_msg	b0_add_traj_table
MSG_B0_BAL_CONFIG	open_dbase	b0_bal_config
MSG_B0_BVOL_ENTRY	download_bvols	b0_bvol_entry
MSG_B0_CANCEL_ROUND	process_a_msg	b0_cancel_round
MSG_B0_CIG_FRAME_RATE	sim_bal_frame_rate	b0_cig_frame_rate
MSG_B0_DATABASE_INFO	open_dbase	b0_database_info
MSG_B0_DELETE_STATIC_VEHICLE	sim_bal_static_remove	b0_delete_static_vehicle
MSG_B0_DELETE_TRAJ_TABLE	db_mcc_setup, process_a_msg	b0_delete_traj_table
MSG_B0_LM_READ	getside	b0_lm_read
MSG_B0_MODEL_DIRECTORY	download_bvols	b0_model_directory
MSG_B0_MODEL_ENTRY	download_bvols	b0_model_entry
MSG_B0_NEW_FRAME	sim_bal_agl_wanted	b0_new_frame
MSG_B0_PROCESS_CHORD	process_a_msg	b0_process_chord
MSG_B0_PROCESS_ROUND	process_a_msg	b0_process_round
MSG_B0_ROUND_FIRED	sim_bal_round_fired, process_a_msg	b0_round_fired
MSG_B0_STATE_CONTROL	sim_bal_start, sim_bal_reset	b0_state_control
MSG_B0_TF_INIT_HDR	db_mcc_setup, process_a_msg	b0_tf_init_hdr
MSG_B0_TF_INIT_PT	db_mcc_setup, process_a_msg	b0_tf_init_pt
MSG_B0_TF_STATE	db_mcc_setup, process_a_msg	b0_tf_state
MSG_B0_TF_VEHICLE_POS	sim_bal_tf_veh_update, process_a_msg	b0_tf_vehicle_pos
MSG_B0_TRAJ_CHORD	sim_bal_traj_chord, sim_bal_req_pt_info, sim_bal_agl_wanted	b0_traj_chord
MSG_B0_TRAJ_ENTRY	db_mcc_setup, process_a_msg	b0_traj_entry

The following table identifies all messages, in alphabetical order, that may be passed from Ballistics to the real-time software. The table also identifies the Ballistics function(s) that push the message onto the Ballistics message queue, and the function in the real-time software that processes it.

The Ballistics functions referenced here are described later in this section. The real-time software functions are described in other sections of this document.

Ballistics --> RTSW Message	Sent By (Ballistics Function)	Processed By (RTSW Function)
MSG_B1_GLOBAL_ADDR	bx_task	none
MSG_B1_HTT_RETURN	b0_new_frame, b0_process_chord, b0_process_round, b0_traj_chord, b0_round_fired	sim_bal_process_msg
MSG_B1_MISS	b0_new_frame, b0_process_chord, b0_process_round, b0_traj_chord, b0_round_fired	sim_bal_process_msg
MSG_B1_ROUND_POSITION	b0_new_frame, b0_process_chord, b0_process_round, b0_round_fired	sim_bal_process_msg
MSG_B1_STATUS_RETURN	bx_task	config_ballistics
MSG_B1_TF_HDR	b0_tf_vehicle_pos	sim_bal_process_msg
MSG_B1_TF_PT	b0_tf_vehicle_pos	sim_bal_process_msg

Ballistics Processing can be divided into the following functional areas:

Ballistics Mainline

- Initializes all Ballistics structures at startup.
- Drives all Ballistics processing.

Ballistics Interface Message Processing

- Processes the Ballistics messages received from the real-time software (usually in response to messages received from the Simulation Host).
- Returns hit, miss, and terrain feedback messages to the real-time software.

Ballistics Database Interaction

- Acquires polygon and bounding volume information from the terrain database and maintains it in a cache using a least-recently used (LRU) swapping algorithm.
- Calculates chord intersections to determine if anything in the simulated environment was hit by a round or trajectory.
- Maintains static vehicles using a set of free lists.

Ballistics Message Queue Management

- Maintains the message queues used as the interface between Ballistics and the real-time software.

Figure 2-8 identifies the CSUs in each functional area of the Ballistics Processing CSC. The CSUs in each area are described in this section, in the order listed above.

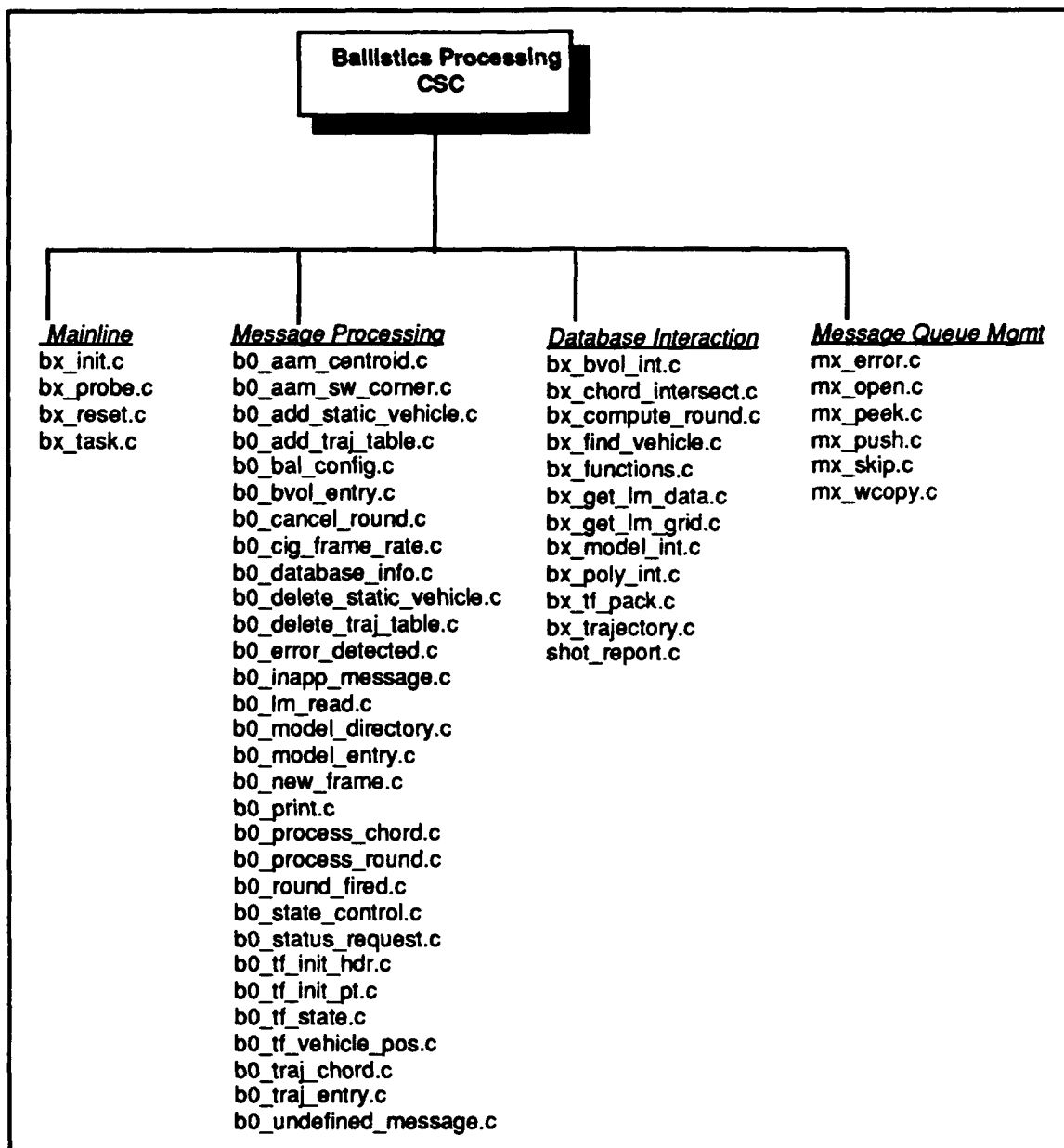


Figure 2-8. Ballistics Processing CSUs

2.4.1 Ballistics Mainline

This section describes the Ballistics Mainline component of Ballistics Processing. The CSUs in this component provide the functions that initialize and drive Ballistics Processing on the CIG.

2.4.1.1 bx_init.c

The bx_init function initializes Ballistics variables and structures at startup.

The function call is **bx_init()**. **bx_init** does the following:

- Initializes and defines the message arrays (**G_init_message[]** and **G_run_message[]**) used by **bx_task** to process incoming messages.
- Initializes the following:
 - Active area memory partition parameters.
 - Dynamic elements database (DED) model directory.
 - DED bounding volume directory.
 - Static vehicle list.
 - Bounding volume cache list.
 - Polygon cache list.
 - Round list.
 - Trajectory table directory.
- Loads the default trajectory tables.
- Initializes various temporary variables.
- Sets the Ballistics state to **BX_INTT**.
- Calls **bx_tf_init_pt_cache** to initialize the terrain feedback point cache (free list).
- Initializes the terrain feedback vehicle list.

Called By: **bx_task**

Routines Called: **bx_tf_init_pt_cache**

Parameters: **none**

Returns: **none**

2.4.1.2 **bx_probe.c**

The **bx_probe** function displays data from the Ballistics database. This function is invoked when the user selects the **b** ("ballistics query menu") option from the Gossip main menu.

The function call is **bx_probe()**. The following table lists the options supported by **bx_probe**, and shows the major steps it performs for each one.

The **PAGE_FORMAT** macro, used by **bx_probe** to handle displays over one page long, is described in Appendix B.

Ballistics Query Menu Option	Processing by bx_probe
? print this menu	Displays options.
A list AAM part info	(not currently implemented)
B list free bvols dir	Displays number of free bvols and location of first.
b list LM bvols	Prompts user for x,y position; uses FIND_LM to get load module number; displays for each bvol: bvol number, type, and next bvol pointer; minimum and maximum x, y, and z; and vertices.
C list bal config	(not currently implemented)
c list terrain corners	(not currently implemented)
F list frame count	(not currently implemented)
i set ballistics addresses	(not currently implemented)
L list LM cache info	Prompts user for x,y position; uses FIND_LM to get load module number; displays cache_flag, frame stamp, poly and bvol counts, max poly z, poly head and tail, max bvol z, bvol head and tail.
l list lm cache	(not currently implemented)
P list free poly dir	Displays number of free polygons and location of first.
p list lm polys	Prompts user for x,y position; uses FIND_LM to get load module number; displays for each polygon: polygon number, next polygon pointer, vertex count, soil type, priority, local terrain flag, shade, grid location, and vertices.
q quit	Exits.
R list active rounds	Displays for each active round: round id, db, type, and mode; tracer type, frame rate, and shot report or proximity; gun position and velocity; azimuth and elevation; frames active, counter, and interval; trajectory entry index, address, size, and offset; start and end.
r list free rounds dir	(not currently implemented)
S list free Stat Veh	Displays number of free static vehicles and location of first.
s list stat vehs	Prompts user for x,y position; uses FIND_LM to get load module number; displays load module number, number of static vehicles, location of first one; displays for each vehicle in load module's list: vehicle id, component count, type, ASID, matrix, next and back pointers.
T list traj directory	Prompts user for table index; displays index, frame rate, effect type, entry count, trajectory address.
t list traj table	Prompts user for trajectory type; displays bx and bz boresight values for each entry.
v find vehicle	Prompts user for vehicle id; calls bx_find_vehicle; outputs coordinates of vehicle's centroid.
x exit	Exits.

Called By: gossip_tick

Routines Called: bx_find_vehicle
FIND_LM
PAGE_FORMAT
printf
scanf
strlen
unbf_getchar

Parameters: none

Returns: none

2.4.1.3 bx_reset.c

The bx_reset function resets Ballistics to an initialized state. This function is called by b0_state_control when the message from the real-time software specifies a new state of BX_RESET.

The function call is bx_reset(). bx_reset does the following:

- Sets G_ballistics_complete to FALSE.
- Frees the memory allocated for the load module lists and trajectory tables.
- Initializes the following:
 - Active area memory partition parameters.
 - Dynamic elements database (DED) model directory.
 - DED bounding volume directory.
 - Static vehicle list.
 - Bounding volume cache list.
 - Polygon cache list.
 - Round list.
 - Trajectory table directory.
 - Various temporary variables.
- Sets the Ballistics state to BX_INIT.
- Calls bx_tf_init_pt_cache to initialize the terrain feedback point cache (free list).
- Initializes the terrain feedback vehicle list.

Called By: b0_state_control

Routines Called: bx_tf_init_pt_cache
free
free133 (if running on a Slave board)

Parameters: none

Returns: none

2.4.1.4 bx_task.c

The bx_task.c CSU contains the main Ballistics task and the function used to deallocate resources when the system is shut down. These functions are:

- bx_task
- bx_task_cleanup

2.4.1.4.1 bx_task

The bx_task function is the main Ballistics task. It is loaded into the task table by the Task Initialization CSC, and is put into the run state by sim_bal_start in the Real-Time Processing CSC.

bx_task does the following:

- Allocates memory for the Ballistics message queues (buffers) used to communicate with the real-time software.
- Calls mpv_addrltg to convert the local addresses for the message queues to global VME addresses.
- Fills the message queues.
- Calls mpv_ncatalog to put the incoming queue's name and address into the MPV catalog.
- Calls bx_init to initialize structures used by Ballistics.
- If running on a Master board:
 - Waits for a message to be posted to the BALLISTICS_MB mailbox.
 - Posts a message to the SIMULATION_MB mailbox.
- Sets pointers to the three message queues.
- Opens the three message queues.
- Notifies the real-time software that Ballistics has started (via a MSG_B1_STATUS_RETURN message).
- If running on a Slave board, gives the real-time software the addresses of Ballistics global variables (via a MSG_B1_GLOBAL_ADDR message).
- Calls poll_shutdown to see if a system shutdown has been requested.
- Calls mx_peek to preview the top message in the incoming message queue.
- If the status returned by mx_peek is MX_DEVICE_EMPTY, calls sc_delay to pause.
- If the status returned by mx_peek is MX_MESSAGE_PREVIEWED:
 - Uses the message code to index into the G_init_message[] or G_run_message[] array, based on the current state of Ballistics. This invokes the appropriate Ballistics Interface Message Processing (b0_*) function to process that message.
 - Calls mx_skip to remove the message from the queue.
 - Previews the next message.

Called By: none (task created and started at initialization time)

Routines Called: *G_init_message[]
 *G_run_message[]
 bx_init
 GLOB
 malloc
 mpv_addr1tg
 mpv_ncatalog
 mx_error
 mx_open
 mx_peek
 mx_push
 mx_skip
 outhexl (if running on a Slave board)
 poll_shutdown
 printf
 puts (if running on a Slave board)
 rt_pend
 rt_post
 sc_delay

Parameters: none

Returns: none

2.4.1.4.2 bx_task_cleanup

The bx_task_cleanup function deallocates the resources owned by bx_task. This function is called if a system shutdown is requested by the Gossip user. It is called via the *task_cleanup function pointer, which points to the cleanup routine's name in the task table.

The function call is **bx_task_cleanup()**.

The function returns 1 if successful, or 0 if an error occurred.

***Note:** This function is not yet implemented. At the current time, it simply returns a 1 if called.*

Called By: poll_shutdown (through *task_cleanup)

Routines Called: none

Parameters: none

Returns: 0
 1

2.4.2 Ballistics Interface Message Processing

This section describes the Ballistics Interface Message Processing component of Ballistics Processing. This component contains the functions that process the Ballistics messages received by `bx_task` from the real-time software.

The Ballistics Interface Message Processing functions are defined as elements of arrays in `bx_init`. Two arrays are used:

G_init_message[]

These messages are used to initialize Ballistics (e.g., define model entries or trajectory tables).

G_run_message[]

These messages are used to respond to runtime messages (e.g., process rounds or manage static vehicles).

The index into either array is the message code (*G_m_code*).

The complete processing mechanism is as follows:

1. The Simulation Host sends a Ballistics message.
2. The real-time software (usually `process_a_msg` or a `sim_bal_*` routine) calls `mx_push` to push the message onto the Ballistics message queue. The calling function sets the *message_code* to `M_B0_<message>`.
3. `bx_task` previews the message from the message queue.
4. `bx_task` indexes into `G_init_message[]` or `G_run_message[]` with the message code (*G_m_code*). (The array used depends on the current Ballistics state.) `bx_task` also passes a pointer to the message (*message_P*).
5. The function corresponding to the specified element in the specified array is called with the *message_P* parameter.

This method of invoking the Ballistics Interface Message Processing functions provides for faster processing than direct function calls.

Note that some of the messages sent from the real-time software to Ballistics do not originate from the Simulation Host. For example, the real-time software generates messages to start and stop Ballistics, and to tell Ballistics where active area memory is. The processing mechanism for such messages is the same as for those received from the Simulation Host.

Some Ballistics messages cause a return message. For example, a `ROUND_FIRED` message results in a `HIT_RETURN` or `MISS` message. The Ballistics Interface Message Processing function generates the response message and calls `mx_push` to push it onto the outgoing message queue with the *message_code* set to `M_B1_<message>`. The real-time software retrieves the message from the queue and processes it accordingly.

For more information on the Ballistics message queues and the `mx_*` functions used to manage them, refer to Section 2.4.4.

2.4.2.1 `b0_aam_centroid.c`

The `b0_aam_centroid` function is a stub for future expansion; it is not currently used.

The function call is `b0_aam_centroid()`. The function always returns 0.

2.4.2.2 `b0_aam_sw_corner.c`

The `b0_aam_sw_corner` function processes the `MSG_B0_AAM_SW_CORNER` message. This message is sent by `sim_bal_start` when Ballistics is first put into the run state. During a simulation, the message is sent by `_rowcol_rd` whenever active area memory is relocated. The message gives Ballistics the coordinates of the new southwest corner of active area memory, and the function calculates the coordinates of the northeast corner.

The function call is `b0_aam_sw_corner(message_P)`, where *message_P* is a pointer to the `MSG_B0_AAM_SW_CORNER` message.

`b0_aam_sw_corner` does the following:

- Sets the global variables `G_terrain_corners.SW_x` and `G_terrain_corners.SW_y` to the values provided in the message.
- Sets `G_terrain_corners.NE_x` by adding twice the viewing range to the `SW_x` value.
- Sets `G_terrain_corners.NE_y` by adding twice the viewing range to the `SW_y` value.

The function always returns 0.

Called By: `bx_task` (through `*G_run_message[]`)

Routines Called: none

Parameters: `MSG_B0_AAM_SW_CORNER` `*message_P`

Returns: 0

2.4.2.3 `b0_add_static_vehicle.c`

The `b0_add_static_vehicle` function processes the `MSG_B0_ADD_STATIC_VEHICLE` message. This message is sent by `sim_bal_static_add` when processing a request by the Simulation Host to add a new static vehicle to the local terrain. Ballistics adds the new vehicle to the tables it uses when determining intersections.

The function call is **b0_add_static_vehicle(message_P)**, where *message_P* is a pointer to the MSG_B0_ADD_STATIC_VEHICLE message.

b0_add_static_vehicle does the following:

- Verifies that the vehicle's load module is within active area memory.
- Verifies that the vehicle's component count is either 1 or 3.
- Uses the NEW_STAT_VEH macro to add the vehicle to the correct load module's vehicle table.
- Copies the vehicle's data from the message to the static vehicle table.
- Sets the cosine and sine of the rotation of the vehicle's second component, if applicable.

The function returns a 0 if successful. It returns 1 if the vehicle's load module is out of range, the maximum vehicle limit has been reached, or the number of components (values used to determine the vehicle's orientation and position) is not 1 or 3.

Called By: bx_task (through *G_run_message[])

Routines Called: BCOPY
 NEW_STAT_VEH

Parameters: MSG_B0_ADD_STATIC_VEHICLE *message_P

Returns: 0
 1

2.4.2.4 **b0_add_traj_table.c**

The **b0_add_traj_table** function processes the MSG_B0_ADD_TRAJ_TABLE message. This message is sent by process_a_msg or db_mcc_setup when processing a MSG_TRAJ_TABLE_XFER message (used to download trajectory tables) from the Simulation Host.

The function call is **b0_add_traj_table(message_P)**, where *message_P* is a pointer to the MSG_B0_ADD_TRAJ_TABLE message.

b0_add_traj_table does the following:

- Verifies that the trajectory type (index) is less than 240.
- Updates the trajectory table pointer with the data from the message.
- For a Master board, frees the trajectory table's entry pointer if it is not NULL.
- Allocates memory for the number of trajectories entries specified in the message.

The function returns 0 if successful, or -1 if the trajectory type is invalid.

Called By: bx_task (through *G_run_message[] and
 *G_init_message[])

Routines Called: free (if running on a Master board)
 MALLOC

Parameters: MSG_B0_ADD_TRAJ_TABLE *message_P

Returns: 0
 -1

2.4.2.5 b0_bal_config.c

The b0_bal_config function processes the MSG_B0_BAL_CONFIG message. This message is sent by open_dbase to give Ballistics its initialized configuration parameters.

The function call is **b0_bal_config(message_P)**, where *message_P* is a pointer to the MSG_B0_BAL_CONFIG message.

b0_bal_config does the following:

- Copies the data from the message to the G_bal_config global variable.
- Sets the global variable G_inv_frame_rate (inverse frame rate) to 1 divided by the frame rate provided in the message.

The function always returns 0.

Called By: bx_task (through *G_init_message[])

Routines Called: BCOPY

Parameters: MSG_B0_BAL_CONFIG *message_P

Returns: 0

2.4.2.6 b0_bvol_entry.c

The b0_bvol_entry function processes the MSG_B0_BVOL_ENTRY message. This message is sent by download_bvols to to add bounding volumes to the DED model directory.

The function call is **b0_bvol_entry(message_P)**, where *message_P* is a pointer to the MSG_B0_BVOL_ENTRY message.

b0_bvol_entry does the following:

- Determines the model directory (terrain or DED).

- For the DED model directory, copies the data from the message to the specified bvol index in the DED bvol directory (G_ded_bvol_dir).
- For the terrain model directory, outputs an error if debug mode is enabled. (The terrain model directory is not currently implemented.)

The function always returns 0.

Called By: bx_task (through *G_init_message[])

Routines Called: BCOPY
 printf (if running on a Master board in debug mode)
 puts (if running on a Slave board in debug mode)

Parameters: MSG_B0_BVOL_ENTRY *message_P

Returns: 0

2.4.2.7 b0_cancel_round.c

The b0_cancel_round function processes the MSG_B0_CANCEL_ROUND message. This message is sent by process_a_msg when the Simulation Host requests that an active round be deleted.

The function call is **b0_cancel_round(message_P)**, where *message_P* is a pointer to the MSG_B0_CANCEL_ROUND message.

b0_cancel_round does the following:

- Searches the active round list for the specified round identifier and type.
- If the round is found, uses the DELETE_ROUND macro to delete the round from the active list and return it to the free list.

The function returns 0 if successful. It returns -1 if the specified round was not found on the active list.

Called By: bx_task (through *G_run_message[])

Routines Called: DELETE_ROUND

Parameters: MSG_B0_CANCEL_ROUND *message_P

Returns: 0
 -1

2.4.2.8 b0_cig_frame_rate.c

The `b0_cig_frame_rate` function processes the `MSG_B0_CIG_FRAME_RATE` message. This message is sent by `sim_bal_frame_rate` to tell Ballistics the current frame rate (10, 15, 30, or 60 Hz).

The function call is `b0_cig_frame_rate(message_P)`, where *message_P* is a pointer to the `MSG_B0_CIG_FRAME_RATE` message.

`b0_cig_frame_rate` does the following:

- Sets the global variable `G_bal_config.cig_frame_rate` to the specified frame rate.
- Sets the global variable `G_inv_frame_rate` (inverse frame rate) to 1 divided by the specified frame rate.

The function always returns 0.

Called By: `bx_task` (through `*G_run_message[]` and `*G_init_message[]`)

Routines Called: none

Parameters: `MSG_B0_CIG_FRAME_RATE` `*message_P`

Returns: 0

2.4.2.9 b0_database_info.c

The `b0_database_info` function processes the `MSG_B0_DATABASE_INFO` message. This message is sent by `open_dbase` after it initializes active area memory partition information. The message specifies the database id (which identifies the backend) and provides information on that backend's active area memory (upper and lower limits, base address, load module size, number of load modules per side, viewing range, etc.).

The function call is `b0_database_info (message_P)`, where *message_P* is a pointer to the `MSG_B0_DATABASE_INFO` message.

`b0_database_info` does the following:

- Allocates space for the load module tables.
- Outputs the location of the load module cache to stdout.
- Loads the load module cache data.
- Allocates memory for and sets up the table of load module addresses (`G_lm_addr`).
- Sets the viewing range (in `G_db_view_range`) for trajectory calculations.

The function always returns 0.

Called By: bx_task (through *G_init_message[])

Routines Called: MALLOC
 printf

Parameters: MSG_B0_DATABASE_INFO *message_P

Returns: 0

2.4.2.10 b0_delete_static_vehicle.c

The `b0_delete_static_vehicle` function processes the `MSG_B0_DELETE_STATIC_VEHICLE` message. This message is sent by `sim_bal_static_rem` when processing a request from the Simulation Host to delete a static vehicle from active area memory.

The function call is `b0_delete_static_vehicle(message_P)`, where *message_P* is a pointer to the `MSG_B0_DELETE_STATIC_VEHICLE` message.

`b0_delete_static_vehicle` does the following:

- Verifies that the vehicle's load module is in active area memory, and that the AAM partition number is valid.
- Searches the specified load module's static vehicle list for the specified vehicle.
- If the vehicle is found, uses the `DELETE_STAT_VEH` macro (described in Appendix B) to remove the vehicle from the load module's list and return it to the free list.

If running on a Slave board and code tracing is enabled, the function outputs data from the message and the static vehicle list to stdout.

The function returns 0 if the static vehicle is successfully deleted. It returns 1 if the vehicle's load module is not in active area memory, or if the AAM partition number is invalid.

Called By: bx_task (through *G_run_message[])

Routines Called: DELETE_STAT_VEH
 outhexl (if running on a Slave board with code_trace)
 puts (if running on a Slave board with code_trace)

Parameters: MSG_B0_DELETE_STATIC_VEHICLE *message_P

Returns: 0
 1

2.4.2.11 b0_delete_traj_table.c

The `b0_delete_traj_table` function is a stub routine for future expansion; it is not currently used.

The function call is `b0_delete_traj_table()`. The function always returns 0.

2.4.2.12 b0_error_detected.c

The `b0_error_detected` function is a stub routine for future expansion; it is not currently used.

The function call is `b0_error_detected()`. The function always returns 0.

2.4.2.13 b0_inapp_message.c

The `b0_inapp_message` function outputs the "*** Inappropriate Message ***" error for Slave boards, and the "*** Inappropriate ballistics message ***" error for Master boards.

The function call is `b0_inapp_message()`. The function always returns 0.

Called By: `bx_task` (through `*G_init_message[]`)

Routines Called: `printf` (if running on a Master board)
 `puts` (if running on a Slave board)

Parameters: none

Returns: 0

2.4.2.14 b0_lm_read.c

The `b0_lm_read` function processes the `MSG_B0_LM_READ` message. This message is sent by `getside` (in `load_modules`) to tell Ballistics that it has added a new load module to the local terrain.

The function call is `b0_lm_read(message_P)`, where `message_P` is a pointer to the `MSG_B0_LM_READ` message.

`b0_lm_read` does the following:

- Sets a pointer to the new load module.
- Uses the `FREE_LM_CACHE` macro (described in Appendix B) to free the new load module's cache.

The function always returns 0.

Called By: bx_task (through *G_run_message[] and
 *G_init_message[])

Routines Called: FREE_LM_CACHE

Parameters: MSG_B0_LM_READ *message_P

Returns: 0

2.4.2.15 b0_model_directory.c

The b0_model_directory function is a stub for future expansion; it is not currently used.

The function call is **b0_model_directory()**. The function always returns 0.

2.4.2.16 b0_model_entry.c

The b0_model_entry function processes the MSG_B0_MODEL_ENTRY message. This message is sent by download_bvols to add entries to the terrain or DED model directory.

The function call is **b0_model_entry(message_P)**, where *message_P* is a pointer to the MSG_B0_MODEL_ENTRY message.

b0_model_entry does the following:

- Determines the model directory (terrain or DED).
- For the DED model directory, copies the data for the entry from the message to the model directory (G_ded_model_dir).
- For the terrain model directory, performs no processing. (The terrain model directory is not currently implemented.)

The function always returns 0.

Called By: bx_task (through *G_init_message[])

Routines Called: BCOPY

Parameters: MSG_B0_MODEL_ENTRY *message_P

Returns: 0

2.4.2.17 b0_new_frame.c

The `b0_new_frame` function processes the `MSG_B0_NEW_FRAME` message. This message is passed by `sim_bal_agl_wanted` (in the Real-Time Processing CSC) at the end of each frame to give Ballistics the frame count and the new state of all dynamic models. `b0_new_frame` then processes each active round.

The function call is `b0_new_frame(message_P)`, where *message_P* is a pointer to the `MSG_B0_NEW_FRAME` message.

For each active round, `b0_new_frame` does the following:

- Calls `bx_trajectory` to see where the round's trajectory ends.
- If the trajectory extends beyond the viewing space:
 - Calls `bx_return_miss` to return a MISS message. (The MISS message is not sent if the round was only to be traced.)
 - Uses the `DELETE_ROUND` macro to delete the round.
- If the trajectory ends within the viewing space:
 - If an intersection with the database is to be computed, calls `bx_find_round_hit`, then uses `DELETE_ROUND` to delete the round.
 - If shot reporting was requested, calls `bx_find_shot_report`.
 - If the round is to be traced, calls `bx_round_tracer_position`.

The function always returns 0.

Called By: `bx_task` (through `*G_run_message[]`)

Routines Called: `bx_find_round_hit`
`bx_find_shot_report`
`bx_return_miss`
`bx_round_tracer_position`
`bx_trajectory`
`DELETE_ROUND`

Parameters: `MSG_B0_NEW_FRAME` `*message_P`

Returns: 0

2.4.2.18 b0_print.c

The `b0_print` function is a generalized message printing routine. The message is printed to `stdout`.

The function call is `b0_print(message_P)`, where *message_P* is a pointer to the message to be printed. The function always returns 0.

Called By:	bx_task	(through *G_run_message[] and *G_init_message[])
Routines Called:	printf puts	(if running on a Master board) (if running on a Slave board)
Parameters:	char	*message_P
Returns:	0	

2.4.2.19 b0_process_chord.c

The `b0_process_chord` function processes the `MSG_B0_PROCESS_CHORD` message. This message is sent by `process_a_msg` when the Simulation Host sends a `MSG_PROCESS_CHORD` message. This message is used to process long chords (e.g., for a laser weapon or intervisibility). Usually, the chord type references a flat trajectory table. The chord is fired using the starting point and the direction angles.

The process chord message from the Simulation Host specifies how the chord is to be processed, as follows:

- The chord's hit or miss is to be computed.
- The chord is to be traced, but not computed for hits or misses.
- The chord has a finite length (the default is infinite).
- The chord is to be processed in one frame or over multiple frames. (Multiple-frame processing is not currently implemented.)
- The chord's intersection is to be calculated but no hit message is to be returned.

The function call is **b0_process_chord(message_P)**, where *message_P* is a pointer to the MSG_B0_PROCESS_CHORD message.

b0_process_chord does the following:

- Gets a pointer to the appropriate trajectory table entry.
- Verifies that the first chord's start point is within active area memory; if not, returns a MISS message.
- Initializes the load module parameters, default effect type, trajectory table pointer, index (count) to trajectory table, start point parameters, and accumulated chord length and test length.
- Computes the square of the trajectory length (specified in the message).
- For each chord in the trajectory, does the following:
 - Checks to see if the trajectory flies beyond the viewing range; if yes, returns a MISS message.
 - Checks to see if the end of the trajectory table has been reached; if yes, returns a MISS message.
 - Rotates through the elevation and azimuth angles.
 - If a finite-length trajectory was specified, checks to see if the chord's length exceeds the length specified in the message; if yes, returns a MISS message.

- Checks to see if the trajectory flies beyond the boundary of active area memory; if yes, returns a MISS message.
- Uses GET_DB_POS to find the load module in which the chord's end point is located.
- Calls bx_chord_intersect to see if the chord hit anything; if yes, returns a HIT_RETURN message.
- If no hit or miss has yet been encountered, sets the next chord's start point equal to the previous chord's end point, and starts over.
- For rounds that are to be traced, calculates the position and returns a ROUND_POSITION message.

The function returns 0 if successful. It returns -1 if there is no trajectory table entry for the specified chord type, the first chord's start point is not within active area memory, or multiple-frame processing was specified.

Called By: bx_task (through *G_run_message[])

Routines Called: bx_chord_intersect
 GET_DB_POS
 GET_LB_FROM_LM
 mx_push
 printf (if running on a Master board in debug mode)
 puts (if running on a Slave board in debug mode)

Parameters: MSG_B0_PROCESS_CHORD *message_P

Returns: 0
 -1

2.4.2.20 b0_process_round.c

The b0_process_round function processes the MSG_B0_PROCESS_ROUND message. This message is sent by process_a_msg when the Simulation Host sends a MSG_PROCESS_ROUND message.

The function call is **b0_process_round(message_P)**, where *message_P* is a pointer to the MSG_B0_PROCESS_ROUND message.

b0_process_round does the following:

- Validates the round type.
- Uses the NEW_ROUND macro (described in Appendix B) to get a round from the free list and put in on the active list.
- Calls bx_guntip_within_db to verify that the gun barrel is within active area memory; if not, uses the DELETE_ROUND macro to delete the round.
- If the message specifies shot reporting, sets shot_report to TRUE and shot_report_done to FALSE.
- Calls bx_trajectory to see if the round's trajectory exceeds active area memory; if yes, returns a MISS message and deletes the round.

- If the trajectory ends within active area memory:
 - If the hit is to be computed, calls `bx_find_round_hit`; then deletes the round.
 - If shot reporting was requested, calls `bx_find_shot_report`.
 - If the round is to be traced, calls `bx_round_tracer_position`.

The function returns 0 if successful. It returns -1 if the round is not of a known type, the free list is empty (i.e., the maximum number of active rounds has been reached), or the gun barrel is not in active area memory.

Called By: `bx_task` (through `*G_run_message[]`)

Routines Called: `bx_find_round_hit`
`bx_find_shot_report`
`bx_guntip_within_db`
`bx_round_tracer_position`
`bx_trajectory`
`DELETE_ROUND`
`mx_push`
`NEW_ROUND`

Parameters: `MSG_B0_PROCESS_ROUND` `*message_P`

Returns: 0
-1

2.4.2.21 `b0_round_fired.c`

The `b0_round_fired` function processes the `MSG_B0_ROUND_FIRED` message. This message is sent by `sim_bal_round_fired` when it receives a `MSG_ROUND_FIRED` message from the Simulation Host.

Note: The MSG_ROUND_FIRED message has been superseded by MSG_PROCESS_ROUND. MSG_ROUND_FIRED is retained for backward compatibility.

The function call is `b0_round_fired(round_fired_P)`, where `round_fired_P` is a pointer to the `MSG_B0_ROUND_FIRED` message.

`b0_round_fired` does the following:

- Validates the round type.
- Calls `NEW_ROUND` to get a round from the free list and put it on the active list.
- Verifies that the gun barrel is within active area memory; deletes the round if it is not.
- Calls `bx_trajectory` to see if the round's trajectory exceeds active area memory; returns a MISS message and deletes the round if it does.
- Calls `bx_chord_intersect` to see what the round hit; returns a HIT_RETURN message and deletes the round.

- For rounds that are to be traced, calculates the position and returns a ROUND_POSITION message.

The function returns 0 if successful. It returns -1 if the round fired is not of a known type, the free list is empty, or the gun barrel is outside active area memory.

Called By: bx_task (through *G_run_message[])

Routines Called: bx_chord_intersect
 bx_trajectory
 DELETE_ROUND
 GET_LB_FROM_LM
 mx_push
 NEW_ROUND

Parameters: MSG_B0_ROUND_FIRED *round_fired_P

Returns: 0
 -1

2.4.2.22 b0_state_control.c

The b0_state_control function processes the MSG_B0_STATE_CONTROL message. This message is sent by sim_bal_start to put Ballistics into the run state, and by sim_bal_reset to reinitialize Ballistics.

The function call is **b0_state_control(message_P)**, where *message_P* is a pointer to the MSG_B0_STATE_CONTROL message.

b0_state_control does the following:

- Sets the Ballistics global variable G_bal_state to the new state provided.
- If the new state is BX_EXIT, sets the Ballistics global variable G_ballistics_complete to TRUE.
- If the new state is BX_RESET, calls bx_reset.
- If running on a Slave board, outputs the new state to stdout.

The function always returns 0.

Called By: bx_task (through *G_run_message[] and
 *G_init_message[])

Routines Called: bx_reset
 outhexl (if running on a Slave board)
 puts (if running on a Slave board)

Parameters: MSG_B0_STATE_CONTROL *message_P

Returns: 0

2.4.2.23 b0_status_request.c

The b0_status_request function is a stub routine for future expansion; it is not currently used.

The function call is **b0_status_request()**. The function always returns 0.

2.4.2.24 b0_tf_init_hdr.c

The b0_tf_init_hdr function processes the MSG_B0_TF_INIT_HDR message. This message is sent by the real-time software when the Simulation Host sends a MSG_TF_INIT_HDR message to initiate terrain feedback reporting for a vehicle. The message specifies the vehicle id, the number of points for which terrain feedback information is to be provided, and the frequency at which feedback data is to be reported (if the message concerns the simulated vehicle). The message is followed by one MSG_TF_INIT_PT message for each point specified in the header message.

The function call is **b0_tf_init_hdr(message_P)**, where *message_P* is a pointer to the MSG_B0_TF_INIT_HDR message.

b0_tf_init_hdr does the following:

- Verifies that the maximum number of terrain feedback vehicles has not been exceeded. (This limit is set in the bx_defines.h file.)
- Verifies that the requested point count is not 0.
- Verifies that the requested point count is not greater than the number of points on the free list (returned by bx_tf_pts).
- Finds a free entry in the vehicle terrain feedback array.
- Adds the vehicle's header entry to the vehicle terrain feedback array.
- Calls bx_tf_new_tf_pts to get the terrain feedback point structures.

The function returns 0 if successful. It returns -1 if the maximum number of vehicles or points has been reached, or if the point count specified in the message is 0.

Called By: bx_task (through *G_run_message[] and
 *G_init_message[])

Routines Called: bx_tf_new_tf_pts
 bx_tf_pts
 printf (in debug mode only)

Parameters: MSG_B0_TF_INIT_HDR *message_P

Returns: 0
 -1

2.4.2.25 b0_tf_init_pt.c

The `b0_tf_init_pt` function processes the `MSG_B0_TF_INIT_PT` message. This message is sent by the real-time software when the Simulation Host sends a `MSG_TF_INIT_PT` message to specify the vehicle coordinates for which terrain feedback information is to be acquired. One `MSG_TF_INIT_PT` message is sent for each point specified in the corresponding `MSG_TF_INIT_HDR` message.

The function call is `b0_tf_init_pt(message_P)`, where *message_P* is a pointer to the `MSG_B0_TF_INIT_PT` message.

`b0_tf_init_pt` does the following:

- Finds the entry for the specified vehicle in the vehicle terrain feedback array.
- Verifies that the vehicle is in use and has a non-zero point count.
- Puts the data for the point in the vehicle's point list.

The function returns 0 if successful. It returns -1 if the specified vehicle is not in the vehicle terrain feedback list, the `_use` flag in the vehicle's header is `FALSE`, or the vehicle's header specifies 0 points.

Called By: **bx_task** (through `*G_run_message[]` and
 `*G_init_message[]`)

Routines Called: **bx_tf_copy_msg**
 bx_tf_next
 printf (in debug mode only)

Parameters: **MSG_B0_TF_INIT_PT** ***message_P**

Returns: 0
 -1

2.4.2.26 b0_tf_state.c

The `b0_tf_state` function processes the `MSG_B0_TF_STATE` message. This message is sent by the real-time software when the Simulation Host sends a `MSG_TF_STATE` message to change terrain feedback processing parameters for a specified vehicle.

The function call is `b0_tf_state(message_P)`, where *message_P* is a pointer to the `MSG_B0_TF_STATE` message.

`b0_tf_state` does the following:

- Finds the vehicle's entry in the vehicle terrain feedback list.

- If the message specifies `BX_TF_CODE_OFF` (disable processing) or `BX_TF_CODE_ON` (enable processing), toggles the processing state in the vehicle's terrain feedback header.
- If the message specifies `BX_TF_CODE_RM` (delete this vehicle from the terrain feedback processing lists):
 - Calls `bx_tf_free_tf_pts` to return the vehicle's terrain feedback points to the free list.
 - Sets the vehicle's `in_use` flag to false.
 - Sets the vehicle's point count to 0.
- If the message specifies `BX_TF_CODE_FREQ` (change the frequency at which terrain feedback messages are sent), sets the vehicle's frame period to the CIG frame rate divided by the frequency specified in the message.

The function returns 0 if successful. It returns -1 if the vehicle's entry cannot be found, its in_use flag is false, or its point count is zero. It also returns -1 if an invalid message code is passed.

Called By: bx_task (through *G_run_message[] and *G_init_message[])

Routines Called: `bx_tf_free_tf_pts`

Parameters: MSG_B0_TF_STATE *message P

Returns: 0
 -1

2.4.2.27 b0_tf_vehicle_pos.c

The `b0_tf_vehicle_pos` function processes the `MSG_B0_TF_VEHICLE_POS` message. This message is sent by the real-time software when the Simulation Host sends a `MSG_TF_VEHICLE_POS` message. This message gives Ballistics the new position and rotation of a vehicle (other than the simulation vehicle) for which terrain feedback data is being collected.

The function call is `b0_tf_vehicle_pos(message_P)`, where *message_P* is a pointer to the MSG_B0_TF_VEHICLE_POS message.

b0_tf_vehicle_pos does the following:

- Finds the vehicle's entry in the vehicle terrain feedback list.
- Builds and returns a MSG_B1_TF_HDR message to the real-time software. This message contains the vehicle's id and point count, and a time stamp.
- For each terrain feedback point assigned to this vehicle:
 - Builds a chord starting at a 20,000-meter elevation.
 - Calls bx_tf_pt_data to get the coordinates of the point.
 - Uses GET_DB_POS to find the load module in which the point resides.
 - Calls bx_chord_intersect to find the chord's intersection with the database.

- If a hit is detected, builds a MSG_B1_TF_PT message with the intersection information and returns it to the real-time software.
- If no hit is detected, resets the chord's starting and ending positions and tries again, for a maximum of five attempts.
- Calls bx_tf_next to get the next point.

The function returns 0 if successful. It returns -1 if the vehicle's entry cannot be found, its in_use flag is false, or its point count is zero.

Called By: bx_task (through *G_run_message[] and *G_init_message[])

Routines Called: bx_chord_intersect
bx_tf_next
bx_tf_pt_data
GET_DB_POS
mx_push
printf (in debug mode only)

Parameters: MSG_B0_TF_VEHICLE_POS *message_P

Returns: 0
-1

2.4.2.28 b0_traj_chord.c

The b0_traj_chord function processes the MSG_B0_TRAJ_CHORD message. This message is sent by sim_bal_traj_chord when processing a MSG_TRAJ_CHORD message, and by sim_bal_req_pt_info when processing a MSG_REQUEST_POINT_INFO message.

The function call is **b0_traj_chord(message_P)**, where *message_P* is a pointer to the MSG_B0_TRAJ_CHORD message.

b0_traj_chord does the following:

- Uses the GET_DB_POS macro (described in Appendix B) to locate the chord's start and end points.
- Calls bx_chord_intersect to determine whether the chord hits anything in the local terrain.
- Pushes a HIT_RETURN or a MISS message (as appropriate) onto the Ballistics outgoing message queue.

The MSG_B0_TRAJ_CHORD message is also sent by the real-time software when processing the simulated vehicle's AGL (altitude above ground level). b0_traj_chord calls bx_chord_intersect to determine what the chord hits, then returns a HIT_RETURN message. If the chord does not intersect, the function outputs information on the AGL chord to stdout.

The function always returns 0.

Called By: bx_task (through *G_run_message[])

Routines Called: bx_chord_intersect
 GET_DB_POS
 mx_push
 outhexl (if running on a Slave board)
 printf
 puts (if running on a Slave board)

Parameters: MSG_B0_TRAJ_CHORD *message_P

Returns: 0

2.4.2.29 b0_traj_entry.c

The b0_traj_entry function processes the MSG_B0_TRAJ_ENTRY message. This message is used to add entries to a trajectory table. The message is sent by the real-time software when processing a MSG_TRAJ_TABLE_XFER message from the Simulation Host.

The function call is **b0_traj_entry(message_P)**, where *message_P* is a pointer to the MSG_B0_TRAJ_ENTRY message.

b0_traj_entry does the following:

- Verifies the trajectory type.
- Finds the specified entry in the trajectory table.
- Puts the boresight x and z values into the table entry.

The function returns 0 if successful, -1 if the trajectory type is invalid, and 1 if the trajectory table is already full.

Called By: bx_task (through *G_run_message[] and
 *G_init_message[])

Routines Called: outhexl (if running on a Slave board in debug mode)
 puts (if running on a Slave board in debug mode)

Parameters: MSG_B0_TRAJ_ENTRY *message_P

Returns: 0
 1
 -1

2.4.2.30 b0_undefined_message.c

The b0_undefined_message function outputs the "*** Undefined Message ***" error to stdout for Slave boards.

The function call is **b0_undefined_message()**. The function always returns 0.

Called By:	bx_task (through *G_run_message[] and *G_init_message[])
Routines Called:	puts (if running on a Slave board)
Parameters:	none
Returns:	0

2.4.3 Ballistics Database Interaction

The Ballistics Database Interaction component of Ballistics Processing is responsible for the following:

- Calculating chord intersections (hits) for various purposes.
- Acquiring polygon and bounding volume information from the terrain database.
- Maintaining polygon and bounding volume information in a cache using an LRU (least-recently-used) swapping algorithm.
- Maintaining static vehicles using a set of free lists.

The driving function in this component is `bx_chord_intersect`. This function is called by the functions in the Ballistics Interface Message Processing component that deal with processing rounds or tracing trajectories. `bx_chord_intersect` calls other Ballistics Database Interaction functions to check for intersections with various objects (static vehicles, dynamic vehicles, terrain bounding volumes, and terrain polygons).

The following points apply to intersection calculations:

- When determining whether a given trajectory intersects with a model or the terrain, Ballistics treats the trajectory as a series of consecutive chords. Each chord is a maximum of 115 meters long. All computations are performed on the chords.
- Intersections with models are calculated using the bounding volume surrounding the model or its articulated part, not with the model itself. A bounding volume, or `bvol`, is the volume of the bounding box that is used to enclose a model in the simulation environment. The use of `bvols` reduces the number of surfaces that Ballistics must deal with. An intersection with any surface of any `bvol` belonging to a model is considered an intersection with that model.
- Intersections with the terrain are calculated with polygons that have the local terrain flag and/or the Ballistics flag set to `TRUE`.

2.4.3.1 `bx_bvol_int.c`

The `bx_bvol_int` function intersects a chord with a bounding volume. This function is called by `bx_chord_intersect` to check for intersections with terrain bounding volumes, and by `bx_model_int` to check for intersections with model (vehicle) bounding volumes.

The function call is `bx_bvol_int(start, end, pbvl, ratio_to_intersect, vehicle_flag)`, where:

start is the chord's starting point

end is a pointer to the return location for the chord's ending point (the intersection point); the value is returned by `bx_bvol_int`

pbvl is a pointer to the `bvol` entry

ratio_to_intersect is a pointer to the return location for the distance from the chord's start point to the intersection point, divided by the total length of the chord; this value is returned by *bx_bvol_int* and is useful when transforming chord points into the vehicle coordinate system

vehicle_flag is **TRUE** if the model is a vehicle, **FALSE** if not

bx_bvol_int does the following:

- Checks the bvol's vertices against the chord's start and end points to see if they intersect.
- Clips backfaces (the sides of a polygon that faces away from the viewpoint).
- Checks for start and end points on the same side of the bounding volume.
- Checks for hits on the top or bottom of the bounding volume.
- Clips around the quadrilateral projection of the bounding volume.
- Sets the chord's ending position.

If an intersection is found, the function returns **TRUE** and places the intersection point and the *ratio_to_intersect* into the locations specified in the call. It returns **FALSE** if no intersection is detected.

Called By: *bx_chord_intersect*
 bx_model_int

Routines Called: none

Parameters:	R4P3D	*start
	R4P3D	*end
	BVOL_ENTRY	*pbvl
	REAL_4	*ratio_to_intersect
	BOOLEAN	vehicle_flag

Returns: 1 (TRUE)
 0 (FALSE)

2.4.3.2 **bx_chord_intersect.c**

The *bx_chord_intersect* function determines whether a given chord intersects with anything in active area memory. It calls other functions in the Ballistics Database Interaction component to check for intersections with models or the terrain, then creates the hit or miss message.

The function call is *bx_chord_intersect(chord_P, buffer_P, aam_index, dv_ex_flag, dv_veh_id)*, where:

chord_P is a pointer to the chord's data

buffer_P is a pointer to the hit return data

aam_index is the active area memory partition index

dv_ex_flag is **TRUE** if a particular vehicle is to be excluded from intersection processing, or **FALSE** if all vehicles are to be included

dv_veh_id is the id of the vehicle to be excluded, if *dv_ex_flag* is **TRUE**

bx_chord_intersect does the following:

- Checks for hits on pre- and post-processed dynamic models.
- Calls *bx_get_lm_grid* to find the load modules to be searched, based on the chord's location.
- Calls *bx_model_int* to check for intersections with static models.
- Calls *bx_model_int* to check for intersections with dynamic models.
- Calls *bx_get_lm_data* to get data for the load module (if not in cache).
- Calls *bx_bvol_int* to check for intersections with terrain bounding volumes.
- Calls *bx_poly_int* to check for intersections with terrain polygons.
- Builds the hit return message (to be returned to the Simulation Host by the calling routine).

The function returns **TRUE** if an intersection is detected. It returns **FALSE** if no intersection was detected, or if the load module could not be found.

Called By: *b0_process_chord*
 b0_round_fired
 b0_tf_vehicle_pos
 b0_traj_chord
 bx_find_round_hit

Routines Called: *BCOPY*
 bx_bvol_int
 bx_get_lm_data
 bx_get_lm_grid
 bx_model_int
 bx_poly_int
 GET_LB_FROM_LM

Parameters:	<i>CHORD</i>	<i>*chord_P</i>
	<i>BYTE</i>	<i>*buffer_P</i>
	<i>WORD</i>	<i>aam_index</i>
	<i>BOOLEAN</i>	<i>dv_ex_flag</i>
	<i>WORD</i>	<i>dv_veh_id</i>

Returns: 1 (**TRUE**)
 0 (**FALSE**)

2.4.3.3 *bx_compute_round.c*

The functions in the *bx_compute_round* CSU are used to process rounds and generate hit or miss messages. These functions are:

- *bx_return_miss*
- *bx_guntip_within_db*
- *bx_find_shot_report*

- `bx_round_tracer_position`
- `bx_find_round_hit`

2.4.3.3.1 `bx_return_miss`

The `bx_return_miss` function builds a `MSG_B1_MISS` message for return to the Simulation Host. This function is called when Ballistics determines that a fired round did not intersect with any object in active area memory.

The function call is `bx_return_miss(message_P, frames_active)`, where:

message_P is a pointer to the `MSG_B0_PROCESS_ROUND` message
frames_active is the number of frames since the round was fired

`bx_return_miss` builds the `MSG_B1_MISS` message, then pushes it onto the outgoing message buffer. The message is generated only if the round was to be computed for intersection (i.e., no miss message is returned for traced rounds).

Called By: `b0_new_frame`
 `bx_guntip_within_db`

Routines Called: `mx_push`

Parameters: `MSG_B0_PROCESS_ROUND` **message_P*
 `HWND` *frames_active*

Returns: `none`

2.4.3.3.2 `bx_guntip_within_db`

The `bx_guntip_within_db` function verifies that the gun barrel is within active area memory. This function is called before the fired round is processed.

The function call is `bx_guntip_within_db(message_P)`, where *message_P* is a pointer to the `MSG_B0_PROCESS_ROUND` message.

`bx_guntip_within_db` does the following:

- Uses the global variable `G_terrain_corners` to determine the four corners of active area memory.
- Compares the gun position's x and y coordinates (from the `MSG_B0_PROCESS_ROUND` message) to the current terrain corners.
- If the gun tip is found to be outside active area memory, calls `bx_return_miss` to generate a `MSG_B1_MISS` message.

The function returns `TRUE` if the gun tip is within active area memory, or `FALSE` if it is not.

Called By: b0_process_round

Routines Called: bx_return_miss

Parameters: MSG_B0_PROCESS_ROUND *message_P

Returns: 1 (TRUE)
 0 (FALSE)

2.4.3.3.3 bx_find_shot_report

The `bx_find_shot_report` function invokes the function that provides feedback on rounds designated for a specific target. This additional information describes how far off the shot was from the intended target's centroid, and in what direction. This function is called for a round if the `MSG_PROCESS_ROUND` message from the Simulation Host specifies shot reporting. The message also specifies the target vehicle's id.

The function call is `bx_find_shot_report(message_P, chord_P, hit_registered, frames_active)`, where:

message_P is a pointer to the `MSG_B0_PROCESS_ROUND` message

chord_P is a pointer to the chord data

hit_registered is `TRUE` if the round hit an object in the database, or `FALSE` if it missed

frames_active is the number of frames since the round was fired

`bx_find_shot_report` does the following:

- Calls `bx_find_vehicle` to get the coordinates of the target vehicle's centroid.
- Calls `shot_report` to determine where the shot intersected the database in relation to the target vehicle's centroid.
- If `shot_report` reports an intersection with the target plane, or a hit is registered, builds a `MSG_B1_SHOT_REPORT` message and pushes it onto the outgoing message buffer.

The function returns the *return_code* returned by `shot_report` if an intersection is detected. It returns `TRUE` if no intersection was reported or if `bx_find_vehicle` could not find the vehicle in active area memory.

Called By: b0_new_frame
 b0_process_round

Routines Called: bx_find_vehicle
 mx_push
 printf (in debug mode only)
 shot_report

Parameters:	MSG_B0_PROCESS_ROUND CHORD BOOLEAN HWORD	*message_P *chord_P hit_registered frames_active
-------------	---	---

Returns:	1 (TRUE) return_code
----------	-------------------------

2.4.3.3.4 bx_round_tracer_position

The `bx_round_tracer_position` function determines the position of a fired round. This function is called if the Simulation Host requests that a round be traced and not intersected with the database. The reply message specifies the starting and ending load modules of the trajectory traced by the fired round.

The function call is `bx_round_tracer_position(message_P, chord_P, round_P)`, where:

message_P is a pointer to the MSG_B0_PROCESS_ROUND message
chord_P is a pointer to the chord data
round_P is a pointer to the round data

`bx_round_tracer_position` does the following:

- Checks to see if load module blocking is enabled.
- Sets the round's starting and ending position based on the chord data; uses the GET_LB_FROM_LM macro if load module blocking is enabled.
- Pushes a MSG_B1_ROUND_POSITION message onto the tracer message buffer.

The function always returns 0.

Called By:	b0_new_frame b0_process_round
------------	----------------------------------

Routines Called:	GET_LB_FROM_LM mx_push
------------------	---------------------------

Parameters:	MSG_B0_PROCESS_ROUND CHORD ROUND_DATA	*message_P *chord_P *round_P
-------------	---	------------------------------------

Returns:	0
----------	---

2.4.3.3.5 **bx_find_round_hit**

The **bx_find_round_hit** function is called to see if a round intersects with anything in the database, and to generate a response message for return to the Simulation Host. This function is called at the end of each frame for each active round. It is also called in response to a **MSG_B0_PROCESS_ROUND** message from the Simulation Host.

The function call is **bx_find_round_hit(message_P, chord_P, round_P, frames_active)**, where:

message_P is a pointer to the **MSG_B0_PROCESS_ROUND** message
chord_P is a pointer to the chord data
round_P is a pointer to the round data
frames_active is the number of frames since the round was fired

bx_find_round_hit does the following:

- Calls **bx_chord_intersect** to see if anything in the local terrain is intersected by the fired round.
- If **bx_chord_intersect** reports a hit:
 - Builds a **MSG_B1_HIT_RETURN** message and pushes it onto the outgoing message buffer.
 - If the request is for a shot report and the shot report has not been done, calls **bx_find_shot_report**.

The function returns the *return_code* returned by **bx_chord_intersect**.

Called By:	b0_new_frame b0_process_round	
Routines Called:	bx_chord_intersect bx_find_shot_report mx_push printf	(in debug mode only)
Parameters:	MSG_B0_PROCESS_ROUND CHORD ROUND_DATA HWORD	*message_P *chord_P *round_P frames_active
Returns:	return_code	

2.4.3.4 **bx_find_vehicle.c**

The **bx_find_vehicle** function finds the coordinates of the centroid of a specified vehicle in active area memory. The centroid is used for shot reporting (determining how close a fired

round came to a target vehicle). This function is also called if the user selects the v ("find vehicle") option from the Ballistics Query menu.

The function call is **bx_find_vehicle(vehicle_id, vehicle_centroid_P)**, where:

vehicle id is the vehicle's identifier

vehicle centroid P is a pointer to the location to store the vehicle's centroid

bx_find_vehicle does the following:

- Looks for the specified vehicle id in the table of dynamic vehicles; if found, places the vehicle's centroid coordinates in the location specified in the call.
- Looks for the specified vehicle id in the table of static vehicles; if found, places the vehicle's centroid coordinates in the location specified in the call.

The function returns `B1_SR_DYNAMIC_VEHICLE` if it finds the vehicle in the dynamic list, and `B1_SR_STATIC_VEHICLE` if it finds the vehicle in the static list. It returns `B1_SR_NONEXISTENT_VEHICLE` if it does not find the vehicle id in either list.

Called By: bx_probe
bx_shot_report

Routines Called: none

Parameters:	HWORD	vehicle_id
	R4P3D	*vehicle centroid P

Returns: 0 (B1_SR_NONEXISTENT_VEHICLE)
2 (B1_SR_DYNAMIC_VEHICLE)
3 (B1_SR_STATIC_VEHICLE)

2.4.3.5 bx_functions.c

The `bx_functions.c` CSU contains utility functions used for Ballistics. These functions are the following:

- `bx_new_round`
- `bx_delete_round`
- `bx_get_db_pos`
- `bx_get_chord_end`
- `bx_new_bvol`
- `bx_free_lm_cache`
- `bx_new_poly`
- `bx_get_lb_from_lm`
- `bx_new_stat_veh`
- `bx_delete_stat_veh`
- `bx_dist_sq_pt_line`

Note: Most of these functions are not currently used. Macros (see Appendix B) are used instead, to increase performance.

2.4.3.5.1 **bx_new_round**

The **bx_new_round** function gets a new round from the free list, and increments the number of active rounds. The function returns a pointer (*new_round_P*) to the new round. The pointer is set to NULL if no free rounds are available.

The function call is **bx_new_round()**.

This function is not currently used. The **NEW_ROUND** macro is used instead.

Called By:	none
Routines Called:	none
Parameters:	none
Returns:	<i>new_round_P</i>

2.4.3.5.2 **bx_delete_round**

The **bx_delete_round** function removes a round from the active list and puts it on the free list. It then decrements the number of active rounds and increments the number of free rounds.

The function call is **bx_delete_round(*dead_round_P*)**, where *dead_round_P* is a pointer to the round to be deleted.

This function is not currently used. The **DELETE_ROUND** macro is used instead.

Called By:	none
Routines Called:	none
Parameters:	ROUND_DATA <i>*dead_round_P</i>
Returns:	none

2.4.3.5.3 bx_get_db_pos

The `bx_get_db_pos` function finds the load module that corresponds to a given point in the database.

The function call is `bx_get_db_pos(point_P, lm_width, inv_lm_width, lm_per_side)`, where:

point_P is a pointer to the location in the database

lm_width is the width of a load module

inv_lm_width is the inverse of the width of a load module

lm_per_side is the number of load modules in a row or column of active area memory (usually 16)

This function is not currently used. The `GET_DB_POS` macro is used instead.

Called By: none

Routines Called: FIND_LM

Parameters:	POINT_DATA	*point_P
	HWND	lm_width
	REAL_4	inv_lm_width
	HWND	lm_per_side

Returns: none

2.4.3.5.4 bx_get_chord_end

The `bx_get_chord_end` function finds the end of the current chord (and, therefore, the beginning of the next chord in the trajectory), given an active round and a trajectory table entry.

The function call is `bx_get_chord_end(chord_P, round_message_P, traj_entry_P, offset)`, where:

chord_P is a pointer to the chord

round_message_P is a pointer to the active round

traj_entry_P is a pointer to the trajectory table entry

offset is the gun barrel velocity offset

This function is not currently used.

Called By: none

Routines Called: none

Parameters: CHORD *chord_P
MSG_B0_PROCESS_ROUND *round_message_P
TRAJ_ENTRY *traj_entry_P
REAL_4 offset

Returns: none

2.4.3.5.5 bx_new_bvol

The `bx_new_bvol` function gets a new bounding volume from the free list and adds it to a load module list. If there are no free bvols, `bx_new_bvol` swaps out the least-recently-used load module. The function returns a pointer (*bvol_P*) to the new bounding volume.

The function call is `bx_new_bvol(lm_dir)`, where *lm_dir* is a load module in the cache.

Called By: bx_get_lm_data

Routines Called: FREE_LM_CACHE

Parameters: LM_CACHE_ENTRY *lm_dir

Returns: bvol_P

2.4.3.5.6 bx_free_lm_cache

The `bx_free_lm_cache` function, when given a load module in the Ballistics database cache, puts the bounding volumes in that module on the free bvol list, and puts the polygons in that module on the free polygon list.

The function call is `bx_free_lm_cache(lm_dir)`, where *lm_dir* is a load module in the cache.

This function is not currently used. The `FREE_LM_CACHE` macro is used instead.

Called By: none

Routines Called: none

Parameters: LM_CACHE_ENTRY *lm_dir

Returns: none

2.4.3.5.7 bx_new_poly

The `bx_new_poly` function gets a new polygon from the free list and puts it on a specified load module list. If there are no free polygons, `bx_new_poly` swaps out the least-recently-used load module. The function returns a pointer (*poly_P*) to the new polygon.

The function call is `bx_new_poly(lm_dir)`, where *lm_dir* is a load module in the cache.

Called By: `bx_get_lm_data`

Routines Called: `FREE_LM_CACHE`

Parameters: `LM_CACHE_ENTRY` **lm_dir*

Returns: *poly_P*

2.4.3.5.8 bx_get_lb_from_lm

The `bx_get_lb_from_lm` function takes a load module number (0 to 1023) and returns the number (0 to 255) of the load block that module is in.

The function call is `bx_get_lb_from_lm(lm)`, where *lm* is the load module number.

This function is not currently used. The `GET_LB_FROM_LM` macro is used instead.

Called By: none

Routines Called: none

Parameters: `INT_4` *lm*

Returns: `row*16 + column`

2.4.3.5.9 bx_new_stat_veh

The `bx_new_stat_veh` function gets a static vehicle from the free list and adds it to the list of the specified load module. The function returns a pointer (*new_sv_P*) to the new static vehicle. It returns NULL if no pointers are available (i.e., the maximum number of static vehicles has been reached).

The function call is **bx_new_stat_veh(veh_table_P)** where *veh_table_P* is a pointer to the vehicle table.

This function is not currently used. The NEW_STAT_VEH macro is used instead.

Called By: none

Routines Called: none

Parameters: STRUCT_P_SV *veh_table_P

Returns: NULL
new_sv_P

2.4.3.5.10 bx_delete_stat_veh

The **bx_delete_stat_veh** function removes a static vehicle from a specified load module list and puts it in the free list.

The function call is **bx_delete_stat_veh(dead_sv_P, table_P)**, where:

dead_sv_P is a pointer to the static vehicle to be deleted
table_P is a pointer to the vehicle table

This function is not currently used. The DELETE_STAT_VEH macro is used instead.

Called By: none

Routines Called: none

Parameters: STAT_VEH *dead_sv_P
STRUCT_P_SV *table_P

Returns: none

2.4.3.5.11 bx_dist_sq_pt_line

The **bx_dist_sq_pt_line** function finds the distance squared between a point and a line segment.

The function call is **bx_dist_sq_pt_line(pt_P, start_P, end_P)**, where:

pt_P is a pointer to the point
start_P is a pointer to the start of the line segment

end_P is a pointer to the end of the line segment

The function returns the result of the calculation as *result*. It returns 1000000.00 if the result is less than 0.

Called By: *bx_model_int*

Routines Called: *none*

Parameters: *R4P3D* **pt_P*
 R4P3D **start_P*
 R4P3D **end_P*

Returns: 1000000.00
 result

2.4.3.6 *bx_get_lm_data.c*

The *bx_get_lm_data* function finds and caches all bounding volumes and polygons in a given load module that have their local terrain or Ballistics bit set to true. The function can also be used to cache all bvols and polygons in the load module, regardless of their local terrain and Ballistics bits. This function is called by *bx_chord_intersect* to get load module data from active area memory if it is not already cached.

The function call is *bx_get_lm_data(lm_addr, lm_dir, poly_mask)*, where:

lm_addr is the address of the load module

lm_dir is the load module directory

poly_mask is **TRUE** if all polygons in the load module are to be cached, or **FALSE** if only polygons with the local terrain or Ballistics bit set are to be cached

bx_get_lm_data does the following:

- Searches the four surrounding grids for polygons.
- Calls *bx_new_poly* to add the polygons found to the load module directory.
- Finds out if there are more polygons/stamps to process.
- Searches the load module for bounding volumes.
- Calls *bx_new_bvol* to add the bvols found to the load module directory.

The function always returns 0.

Called By: *bx_chord_intersect*

Routines Called: *bx_new_bvol*
 bx_new_poly
 FXTO881

Parameters:	WORD	lm_addr
	LM_CACHE_ENTRY	*lm_dir
	WORD	poly_mask

Returns: 0

2.4.3.7 bx_get_lm_grid.c

The `bx_get_lm_grid` function finds the load modules and grids in the database that are intersected by a given chord. This function is called by `bx_chord_intersect` when it is looking for the load modules to search.

The function call is `bx_get_lm_grid(pcrd, lm_per_side, bal_search, dvl_search, lm_width, lm_addr_table)`, where:

pcrd is a pointer to the chord
lm_per_side is the number of load modules in a row or column of active area memory
bal_search is used to store load module offsets and grid words
dvl_search is used to store dynamic module path information
lm_width is the width of a load module
lm_addr_table is an array of load module addresses

The function returns TRUE if successful. It returns FALSE if the chord crosses four load modules, but one of the grids is not a corner grid of a load module; this is an error condition.

Called By: bx_chord_intersect

Routines Called: none

Parameters:	CHORD	*pcrd
	HWND	lm_per_side
	LM_SEARCH_LIST	bal_search[]
	HWND	dvl_search[]
	HWND	lm_width
	WORD	lm_addr_table[]

Returns: 1 (TRUE)
0 (FALSE)

2.4.3.8 bx_model_int.c

The `bx_model_int` function intersects a chord with a model. This function is called by `bx_chord_intersect` to check for intersections with static and dynamic vehicles.

The function call is `bx_model_int(chord_P, model_inst_P, hit_data_P)`, where:

chord_P is a pointer to the chord
model_inst_P is a pointer to the model
hit_data_P is a pointer to the data for the hit return message

bx_model_int does the following:

- Based on the model's radius, checks to see if the chord falls completely outside of the model. Returns FALSE if it does.
- Checks the model's first component for a hit.
 - Converts the chord to vehicle coordinates.
 - Translates and rotates the chord.
 - Calls **bx_bvol_int** to check for a bounding volume intersection.
 - If an intersection is found:
 - * Sets *hit_flag* to TRUE.
 - * Uses the *ratio_to_intersect* to determine the coordinates of the intersection point. For the x and y coordinates, subtracts a fixed percentage (**INTERSECT_OFFSET**) from the *ratio_to_intersect* value. This moves the intersection point slightly away from the middle of the object enclosed by the intersected bvol, causing any special effects for the hit to appear largely outside of the object.
 - * Places the hit information in *hit_data_P*.
- If no hit was found, checks the model's second component, if it has one.
 - Rotates the chord into turret coordinates.
 - Calls **bx_bvol_int** to check for a bounding volume intersection. If an intersection is found, sets *hit_flag* to TRUE, applies the *ratio_to_intersect* value as explained above, and places the hit information in *hit_data_P*.

The function returns TRUE if a hit is detected, or FALSE if no intersection is detected.

Called By: **bx_chord_intersect**

Routines Called: **bx_bvol_int**

Parameters:	CHORD STAT_VEH MSG_B1_HIT_RETURN	*chord_P *model_inst_P *hit_data_P
-------------	--	--

Returns:	FALSE hit_flag
----------	-------------------

2.4.3.9 **bx_poly_int.c**

The **bx_poly_int** function intersects a chord and a polygon. This function is called by **bx_chord_intersect** to check for intersections with terrain polygons.

The function call is **bx_poly_int(start, end, vtx_count, poly_P)**, where:

start is the starting point of the chord

end is a pointer to the return location for the ending point of the chord (the point of intersection)

vtx_count is the number of vertices in the polygon

poly_P is a pointer to the polygon's cache entry

bx_poly_int does the following:

- Clips around the polygon using the minimum and maximum values and a fixed offset.
- Makes the polygon normals.
- Calculates the cross product.
- Clips out backface intersections.
- Checks to see if the intersection is in the interior of the polygon.
- Finds the normal-to-polygon side by taking the cross product of the polygon normal and the polygon side.

The function returns TRUE if the chord intersects the polygon, or FALSE if it does not. If an intersection is detected, the intersection point is placed in the *end* location specified in the call.

Called By: *bx_chord_intersect*

Routines Called: none

Parameters:	WORD	<i>vtx_count</i>
	R4P3D	<i>*start</i>
	R4P3D	<i>*end</i>
	POLY_CACHE_ENTRY	<i>*poly_P</i>

Returns: 1 (TRUE)
 0 (FALSE)

2.4.3.10 *bx_tf_pack.c*

The *bx_tf_pack.c* CSU contains utility functions used to process terrain feedback points. These functions are:

- *bx_tf_init_pt_cache*
- *bx_tf_pts*
- *bx_tf_next*
- *bx_tf_copy_msg*
- *bx_tf_pt_data*
- *bx_tf_new_tf_pts*
- *bx_tf_free_tf_pts*

2.4.3.10.1 **bx_tf_init_pt_cache**

The **bx_tf_init_pt_cache** function initializes the terrain feedback points cache (free list). The size of the list is **MAX_TF_PT**, which is defined in **bx_defines.h**. This function is called when Ballistics is initialized or reset.

The function call is **bx_tf_init_pt_cache()**.

Called By: **bx_init**
 bx_reset

Routines Called: **none**

Parameters: **none**

Returns: **none**

2.4.3.10.2 **bx_tf_pts**

The **bx_tf_pts** function returns the count of terrain feedback points in the free list. (These are feedback points that are not currently assigned to a vehicle.) This function is used by **b0_tf_init_hdr** to make sure there are enough free points for a new terrain feedback vehicle.

The function call is **bx_tf_pts()**.

Called By: **b0_tf_init_hdr**

Routines Called: **none**

Parameters: **none**

Returns: **free_tf_pt_P.count**

2.4.3.10.3 **bx_tf_next**

The **bx_tf_next** function, given a terrain feedback point, returns a pointer to the next feedback point for the same vehicle.

The function call is **bx_tf_next(tf_pt_P)**, where **tf_pt_P** is a pointer to the terrain feedback point.

Called By: b0_tf_init_pt
 b0_tf_vehicle_pos

Routines Called: none

Parameters: TF_PT *tf_pt_P

Returns: tf_pt_P->next_P

2.4.3.10.4 bx_tf_copy_msg

The `bx_tf_copy_msg` function places a pointer to a terrain feedback message (`MSG_B0_TF_INIT_PT`) into a point's entry in the terrain feedback list.

The function call is `bx_tf_copy_msg(message_P, tf_pt_P)`, where:

message_P is a pointer to the `MSG_B0_TF_INIT_PT` message
tf_pt_P is a pointer to the terrain feedback point

Called By: b0_tf_init_pt

Routines Called: none

Parameters: MSG_B0_TF_INIT_PT *message_P
 TF_PT *tf_pt_P

Returns: none

2.4.3.10.5 bx_tf_pt_data

The `bx_tf_pt_data` function returns the data (message) associated with a specified terrain feedback point.

The function call is `bx_tf_pt_data(tf_pt_P)`, where *tf_pt_P* is a pointer to a terrain feedback point.

Called By: b0_tf_vehicle_pos

Routines Called: none

Parameters: TF_PT *tf_pt_P

Returns: `&tf_pt_P->pt`

2.4.3.10.6 bx_tf_new_tf_pts

The `bx_tf_new_tf_pts` function allocates new terrain feedback points from the free list to a vehicle. This function is used by `b0_tf_init_hdr` to get the feedback point structures for a new terrain feedback vehicle.

The function call is `bx_tf_new_tf_pts(point_total, tail_PP)`, where:

point_total is the number of feedback points to be allocated

tail \overline{P} is a pointer to the last entry point allocated

bx_tf_new_tf_pts verifies that the free list has enough points available to satisfy the request, then deletes the points from the free list.

The function returns a pointer to the first feedback point allocated to the vehicle if successful. It returns NULL if the free list does not contain enough points.

Called By: b0_tf_init_hdr

Routines Called: none

Parameters:	INT_2	point_total
	TF PT	**tail PP

Returns: tf_pt_P

2.4.3.10.7 bx tf free tf pts

The `bx_tf_free_tf_pts` function returns terrain feedback points to the free list. This function is called by `b0_tf_state` when the Simulation Host sends a message requesting that feedback points be removed from a specified vehicle.

The function call is **bx tf free tf pts(head P, tail P, point total)**, where:

head P is a pointer to the first point to be freed

tail \bar{P} is a pointer to the last point to be freed

point total is the number of feedback points to be freed

bx_tf_free_tf_pts sets the next pointer (**next_P**) for the last entry in the free list to the first point (**head_P**) specified in the message. It also increments the free point count by the point total specified in the message.

Called By: b0_tf_state

Routines Called: none

Parameters: TF_PT *head_P
TF_PT *tail_P
INT_2 point_total

Returns: none

2.4.3.11 bx_trajectory.c

The **bx_trajectory** function returns the position of a projectile using the provided trajectory tables.

The function call is **bx_trajectory(round_P)**, where *round_P* is a point to the round data. **bx_trajectory** does the following:

- If this is the first call for a new round, finds the trajectory table for the round type.
- Rotates through the elevation angle.
- Rotates through the azimuth angle.
- Adds in the gun position and velocity.

The function returns TRUE if it finds the position in the database. It returns FALSE if the round travels beyond the viewing space, or if the end of the trajectory table was reached.

Called By: b0_new_frame
b0_process_round
b0_round_fired

Routines Called: GET_DB_POS

Parameters: ROUND_DATA *round_P

Returns: 1 (TRUE)
0 (FALSE)

2.4.3.12 shot_report.c

The **shot_report** function provides feedback to the Simulation Host on shots that are designated for a specific target. The target's vehicle id is specified in the **MSG_PROCESS_ROUND** message. **shot_report** determines the distance between the shot's intersection and the target vehicle's centroid, and the direction in which the shot was off center.

To determine where a shot intersected in relation to the target vehicle, the function intersects the chord defined by the shot with a target plane. The target plane is defined as a vertical

plane with its origin at the target's centroid and with a normal defined by the x,y projection of the Ballistics chord. The target plane x,z coordinates form a right-hand system with a positive y-axis having the same direction as the projected chord.

The function call is **shot_report(chord_start, chord_end, target_centroid, x_offset, z_offset)**, where:

chord_start is the starting point of the Ballistics chord
chord_end is the ending point of the Ballistics chord
target_centroid is the origin of the target plane (the target vehicle's centroid)
x_offset is a pointer to the x coordinate of the intersection offset; this value is calculated by shot_report
z_offset is a pointer to the z coordinate of the intersection offset; this value is calculated by shot_report

shot_report does the following:

- Gets the normal and vector from the centroid to the chord end.
- Checks to see if the chord intersects the plane.
- If the chord meets the plane, finds the intersection of the normal with the plane.
- Gets the target plane coordinates of the intersection point, and places them in the locations specified in the call.
 - The *x_offset* is the distance (signed) from the centroid to the intersection point in the world x,y plane.
 - The *z_offset* is the world z offset of the intersection point from the centroid's z coordinate.

The function returns TRUE if the chord intersects the plane, or FALSE if no intersection is detected. (A FALSE value usually indicates that the shot intersected an object before reaching the target vehicle.)

Called By: bx_find_shot_report

Routines Called: sqrt

Parameters:	R4P3D R4P3D R4P3D REAL_4 REAL_4	*chord_start *chord_end *target_centroid *x_offset *z_offset
-------------	---	--

Returns: 1 (TRUE)
 0 (FALSE)

2.4.4 Ballistics Message Queue Management

This section details the CSUs in the Ballistics Message Queue Management component of Ballistics Processing. These functions are responsible for manipulating and maintaining the queues that make up the interface between Ballistics and real-time software.

Three message queues are used:

G_indev_P (incoming)

Used for incoming messages (passed to Ballistics). Various functions in the real-time software push messages onto this queue. `bx_task` previews each message and calls the appropriate `b0_*` function to process it.

G_outdev_P (outgoing)

Used for outgoing messages (passed from Ballistics). Various Ballistics functions push messages onto these queues and the real-time software (usually the `sim_bal_process_msg` function in the Real-Time Processing CSC) retrieves them.

G_tracerdev_P (tracer)

Used to report the round position for traced trajectories. Various Ballistics functions push the tracer messages onto this queue and the real-time software (`sim_bal_process_tracer` in the Real-Time Processing CSC) retrieves them.

The structure of each queue is the same. The typedef is provided in the `mx_defines.h` file.

2.4.4.1 `mx_error.c`

The `mx_error` function returns a text message for output to the operator. Messages are provided for the various errors generated when processing messages (`MX_DEVICE_TABLE_FULL`, `MX_DEVICE_BUSY`, etc.) as well as for normal processing states (`MX_MESSAGE_PUSHED`, `MX_MESSAGE_POPPED`, etc.).

This function is called by `bx_task` to display a message if an error occurs when it tries to preview the top message in the message queue. It is called by some of the real-time software functions if an error occurs when the function attempts to push a message onto the Ballistics message queue.

The function call is `mx_error(status)`, where *status* is the current MX state.

Called By:	<code>bx_task</code> <code>download_bvols</code> <code>sim_bal_static_add</code> <code>sim_bal_static_rem</code>
------------	---

Routines Called:	none
------------------	------

Parameters:	WORD	status
-------------	------	--------

Returns: "DEVICE CLOSED"
"DEVICE TABLE FULL"
"DEVICE OPENED"
"DEVICE BUSY"
"DEVICE EMPTY"
"DEVICE FULL"
"MESSAGE PUSHED"
"MESSAGE POPPED"
"MESSAGE PREVIEWED"
"MESSAGE SKIPPED"
"UNDEFINED ERROR"
"UNDEFINED RETURN"

2.4.4.2 **mx_open.c**

The `mx_open` function opens an MX queue device. This function is called by `bx_task` to open the three message queues at startup.

The function call is `mx_open(dev_P, device_size)`, where:

dev_P is a pointer to the MX device (message queue)
device_size is the size of the message queue

The function always returns `MX_DEVICE_OPENED`.

Called By:	<code>bx_task</code>	
Routines Called:	<code>sc_lock</code> <code>sc_unlock</code>	
Parameters:	<code>MX_DEVICE</code> <code>INT_4</code>	<code>*dev_P</code> <code>device_size</code>
Returns:	<code>MX_DEVICE_OPENED</code>	

2.4.4.3 **mx_peek.c**

The `mx_peek` function previews the message at the head of a specified queue. This function is used by `bx_task` to determine what type of message is in the incoming queue, so it can call the appropriate Interface Message Processing (`b0_*`) function to process it. It is also used by various functions in the real-time software to read the messages returned from Ballistics in the outgoing and tracer queues.

The function call is `mx_peek(dev_P, message_code, message_size, message_addr)`, where:

dev_P is a pointer to the message queue

message_code is the message type
message_size is the size of the message in bytes
message_addr is a pointer to the message's address

mx_peek does the following:

- Locks the queue.
- Checks to see if the specified queue is empty.
- Sets a pointer to the first message in the queue.
- Places the message's type and size in *message_code* and *message_size*.
- If the message code is **MX_SKIP**, starts over with the next message in the queue.
- Places a pointer to the message in *message_addr*.
- Unlocks the queue.

The function returns **MX_MESSAGE_PREVIEWED** if successful. It returns **MX_DEVICE_EMPTY** if the specified queue contains no messages.

Called By: **bx_task**
 config_ballistics
 sim_bal_process_msg
 sim_bal_process_tracer

Routines Called: **sc_lock**
 sc_unlock

Parameters: **MX_DEVICE** ***dev_P**
 HWND ***message_code**
 HWND ***message_size**
 BYTE ****message_addr**

Returns: **MX_DEVICE_EMPTY**
 MX_MESSAGE_PREVIEWED

2.4.4.4 **mx_push.c**

The **mx_push** function pushes a message onto a Ballistics message queue. This function is used by various Ballistics functions to add messages to the outgoing and tracer queues, and by various real-time software functions to add messages to the incoming queue.

The function call is **mx_push(dev_P, source_address, message_code, message_size)**, where:

dev_P is a pointer to the message queue
source_address is the address of the message
message_code is the type of message
message_size is the number of bytes in the message

mx_push does the following:

- Locks the queue.
- Verifies that there is room in the queue for the message.
- Copies the message to the end of the queue
- Unlocks the queue.

The function returns `MX_MESSAGE_PUSHED` if successful. It returns `MX_DEVICE_FULL` if the specified message queue is already full.

Called By: `_rowcol_rd`
 `b0_new_frame`
 `b0_process_chord`
 `b0_process_round`
 `b0_round_fired`
 `b0_tf_vehicle_pos`
 `b0_traj_chord`
 `bx_find_round_hit`
 `bx_find_shot_report`
 `bx_return_miss`
 `bx_round_tracer_position`
 `bx_task`
 `db_mcc_setup`
 `download_bvols`
 `getside`
 `open_dbase`
 `process_a_msg`
 `sim_bal_agl_wanted`
 `sim_bal_frame_rate`
 `sim_bal_req_pt_info`
 `sim_bal_reset`
 `sim_bal_round_fired`
 `sim_bal_start`
 `sim_bal_static_add`
 `sim_bal_static_rem`
 `sim_bal_tf_veh_update`
 `sim_bal_traj_chord`

Routines Called: `BCOPY`
 `sc_lock`
 `sc_unlock`

Parameters:	<code>MX_DEVICE</code>	<code>*dev_P</code>
	<code>WORD</code>	<code>source_address</code>
	<code>HWORD</code>	<code>message_code</code>
	<code>HWORD</code>	<code>message_size</code>

Returns: `MX_DEVICE_FULL`
 `MX_MESSAGE_PUSHED`

2.4.4.5 **mx_skip.c**

The **mx_skip** function skips over a message in the queue. The message at the head of the queue is flushed, and the next message moves to the top of the queue. This function is used to remove messages from a queue after they have been previewed and processed.

The function call is **mx_skip(dev_P)**, where *dev_P* is a pointer to the queue.

The function returns **MX_MESSAGE_SKIPPED** if successful. It returns **MX_DEVICE_EMPTY** if the specified message queue contains no messages.

Called By: **bx_task**
 config_ballistics
 sim_bal_process_msg
 sim_bal_process_tracer

Routines Called: **sc_lock**
 sc_unlock

Parameters: **MX_DEVICE** ***dev_P**

Returns: **MX_DEVICE_EMPTY**
 MX_MESSAGE_SKIPPED

2.4.4.6 **mx_wcopy.c**

The **mx_wcopy** function performs a block copy. This function is used to copy messages.

The function call is **mx_wcopy (source_P, destination_P, byte_count)**, where:

source_P is a pointer to the source data
destination_P is a pointer to the destination location
byte_count is the number of bytes to be copied

This function is not currently used.

Called By: none

Routines Called: none

Parameters: **WORD** ***source_P**
 WORD ***destination_P**
 INT_2 **byte_count**

Returns: none

2.5 CIG Configuration (/cig/libsrc/libconfig)

The functions in the CIG Configuration CSC are involved primarily with preparing the CIG to run a simulation. These functions do the following:

- Initialize and allocate active area memory.
- Initialize the viewport configuration tree.
- Set up calibration, gunner, and gun barrel overlays for T backends. (These are hard-coded overlays that can be displayed on a viewport on top of the terrain display.)
- Maintain the structure that contains the simulated vehicle's current position.

Figure 2-9 identifies the CSUs in the CIG Configuration CSC. These CSUs are described in this section.

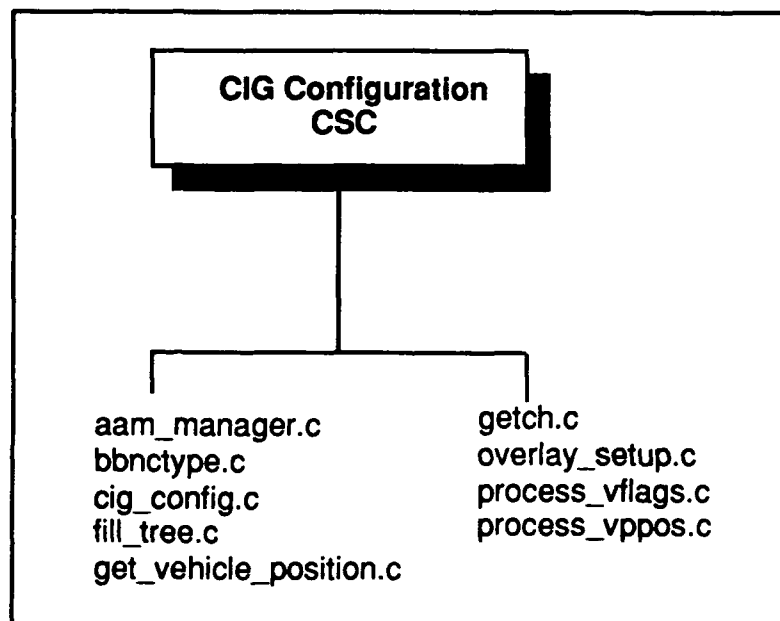


Figure 2-9. CIG Configuration CSUs

2.5.1 aam_manager.c

The functions in aam_manager.c are used to allocate and manage the system (static) and dynamic areas of active area memory. Dynamic memory is located in the double-buffer area; static memory is not double-buffered.

The functions in aam_manager.c are:

- aam_malloc

- return_aam_ptr
- system_aam_init
- dynamic_aam_init

2.5.1.1 aam_malloc

The aam_malloc function allocates system and dynamic active area memory for viewport configuration nodes, cloud models, and gun overlays.

The function call is **aam_malloc(static_flag, num_of_bytes)**, where:

static_flag identifies the area of memory (SYSTEM or DYNAMIC)

num_of_bytes is the number of bytes of memory requested

aam_malloc does the following:

- Determines which area of memory is being requested.
- Verifies that sufficient memory is available.
- Allocates the memory and returns a pointer (*temp_ptr*) to it.

If there is insufficient memory to process the request, aam_malloc returns NULL and displays the amount of memory available.

Called By: cig_config
 cloud_init
 make_m1_overlays
 make_m2_overlays
 vpt_cnode_set_matrix
 vpt_path_init
 vpt_root_init
 vpt_vpt_set

Routines Called: printf

Parameters: BYTE static_flag
 WORD num_of_bytes

Returns: temp_ptr
 NULL

2.5.1.2 return_aam_ptr

The return_aam_ptr function returns the address of the next available location in the static or dynamic area of active area memory.

The function call is **return_aam_ptr(static_flag)**, where *static_flag* identifies the area of memory (SYSTEM or DYNAMIC).

return_aam_ptr returns *system_aam_limit* (the next available address in static memory) or *dynamic_aam* (the next available address in dynamic memory).

Called By: _copy_reconfigurable_viewports_section
_update_second_active_area_memory
cig_config

Routines Called: none

Parameters: **BYTE** **static_flag**

Returns: `system_aam_limit`
 `dynamic_aam`

2.5.1.3 system_aam_init

The `system_aam_init` function initializes the system (static) section of active area memory.

The function call is **system aam init(system aam addr, limit)**, where:

system_aam_addr is the starting address of the memory to be initialized
limit is the ending address of the memory to be initialized

The function returns the starting address of the initialized memory (the address specified by *system aam addr*) as *system aam*.

Called By: cig_config

Routines Called: `printf` (in debug mode only)

Parameters:	WORD	system_aam_addr
	WORD	limit

Returns: `system_aam`

2.5.1.4 `dynamic_aam_init`

The `dynamic_aam_init` function initializes the dynamic section of active area memory. The function call is `dynamic_aam_init(dynamic_aam_addr, limit)`, where:

dynamic_aam_addr is the starting address of the memory to be initialized
limit is the ending address of the memory to be initialized

The function returns the starting address of the initialized memory (the address specified by *dynamic_aam_addr*) as *dynamic_aam*.

Called By: *cig_config*

Routines Called: *printf* (in debug mode only)

Parameters: WORD *dynamic_aam_addr*
 WORD *limit*

Returns: *dynamic_aam*

2.5.2 *bbnctype.c*

bbnctype is a runtime library that defines control characters, punctuation, digits, and alphas.

This file is not currently used.

Called By: none

Routines Called: none

Parameters: none

Returns: none

2.5.3 *cig_config.c*

The *cig_config* function is the message handler for the Viewport Configuration process. It is responsible for processing messages from the Simulation Host to build the configuration tree before runtime. *cig_config* is called by *db_mcc_setup* (in the Real-Time Processing CSC) when the CIG Control message from the Simulation Host specifies *C_CIG_CONFIG*.

The function call is *cig_config(state)*, where *state* is the current state of the CIG system (*C_CIG_CONFIG*).

cig_config does the following:

- Calls *vpt_init_mode_on* to set the viewport initialization mode variable to TRUE. This indicates that viewport configuration is in progress.
- Calls *dynamic_aam_init* to initialize and set up a pointer to the dynamic area of active area memory.

- Calls `system_aam_init` to initialize and set up a pointer to the system (static) area of active area memory.
- Calls `aam_malloc` to allocate memory for the view mode words.
- Calls `aam_malloc` to allocate memory for the daylight TV thermal word (`dtv_therm_word`).
- Loads the reconfigurable data that goes into double-buffered active area memory into DB0.
- Calls `make_cal_overlay` to create the calibration overlay.
- Calls `cloud_init` to initialize the top and bottom cloud models.
- Initializes various configuration variables (`agl_wanted`, `thermal_mode_wanted`, `terrain_feedback_wanted`, and `clouds_wanted_G`) to FALSE.
- For each configuration message received from the Simulation Host:
 - Increments the packet size; calls `syserr` to generate an error message if the packet length specified in the packet header has been exceeded.
 - Calls `cigsimio_msg_in` to write the message to a buffer (if debug message printing or recording has been enabled).
 - Processes the message (see table below).

The following table summarizes the processing performed by `cig_config` in response to each valid message type it receives from the Simulation Host. The first column lists the messages in alphabetical order. The second column briefly describes the purpose of the message (in italics), then lists the major steps performed by `cig_config` to process the message.

If an unknown message type is received, `cig_config` calls `syserr` to generate an "illegal message type" error message.

Message from SIM Host	Processing by cig_config
MSG_AGL_SETUP	<i>Toggles AGL processing on/off.</i> Sets agl_wanted variable to new state requested.
MSG_AMMO_DEFINE	<i>Define ammunition maps.</i> Sets values in ammo_map array.
MSG_CIG_CTL C_NULL C_STOP	<i>Causes a transition to another performance state.</i> No action. Calls fill_tree; calls trav_tree; calls return_aam_ptr; calls dtp_compiler; calls vpt_init_mode_off; copies reconfigurable viewport data from DB0 to DB1; outputs a message if local terrain or terrain feedback processing is enabled; calls flagoff to turn off all viewport configuration debug flags; returns to db_mcc_setup.
MSG_CREATE_CONFIGNODE	<i>Creates a configuration tree node entry.</i> Calls vpt_cnode_process; outputs error if return status is not SUCCEED.
MSG_DEFINE_TX_MODE	<i>Sets up resolution modes for TX backends.</i> Calls mpvideo_define_mode.
MSG_DR11_PKT_SIZE	<i>Specifies exchange packet parameters.</i> Sets incoming and outgoing exchange packet size, local terrain chunk size, local terrain message interval, and CIG hardware type.
MSG_END	<i>Signals end of packet buffer.</i> Calls cigsimio_msg_out to write message to buffer (if message display or recording is enabled); calls start_watch; calls the appropriate data exchange routine (using *exchange_data) to send output and receive input buffers.
MSG_LT_STATE	<i>Changes the local terrain message interval or chunk size.</i> Calls lt_state.
MSG_OVERLAY_SETUP	<i>Places overlays on specified viewports.</i> Calls overlay_setup.
MSG_VIEW_FLAGS	<i>Sets system view flags (on/off, daylight/TV, etc.).</i> Outputs error (this message type is no longer handled by cig_config).
MSG_VIEWPORT_STATE	<i>Defines all viewport parameters.</i> Calls vpt_vpt_process; outputs error if returned status is not SUCCEED.

Called By: db_mcc_setup

Routines Called:

- *exchange_data
- aam_malloc
- bcopy
- cigsimio_msg_in
- cigsimio_msg_out
- cloud_init
- dtp_compiler
- dynamic_aam_init

fill_tree
flagoff
GLOB
lt_state
make_cal_overlay
mpvideo_define_mode
overlay_setup
printf
return_aam_ptr
start_watch
syserr
system_aam_init
trav_tree
vpt_cnode_process
vpt_init_mode_off
vpt_init_mode_on
vpt_vpt_process
vpti_state_dtproc (in debug mode only)
vpti_state_edtpc (in debug mode only)
vpti_state_efillt (in debug mode only)

Parameters: INT_2 state

Returns: `none`

2.5.4 fill_tree.c

The functions in the `fill_tree.c` CSU are responsible for setting the graphics path flags in the configuration tree. These functions are:

- `fill_tree`
- `power`

2.5.4.1 fill tree

The `fill_tree` function sets the graphics path flags in configuration tree nodes. This function is called after all configuration node messages have been processed.

The function call is `fill_tree()`. `fill_tree` does the following:

- Calls `vpti_get_ptr_path` to get a pointer to the first graphics path parameters.
- Uses the graphics path entry path id to set a bit in the configuration node path flag. For example, a path flag of 0x0021 means the node contains information valid for graphics paths 0 and 5.
- Calls `vpti_get_ptr_path` to get the next pointer. Traverses up the configuration tree, setting the path flags in the configuration nodes.

Called By: `cig_config`

Routines Called: power
 printf
 vpti_get_ptr_path
 vpti_state_dtproc (in debug mode only)

Parameters: none

Returns: none

2.5.4.2 power

The power function raises a base to a power. This function is called by fill_tree when it traverses the configuration tree.

The function call is **power(base, n)**, where:

base is the base to be raised
n is the power

The calculated value is returned as *result*.

Called By: fill_tree

Routines Called: none

Parameters: INT_4 base
 INT_4 n

Returns: result

2.5.5 get_vehicle_position.c

The get_vehicle_position function is not used by the standard GT100 system.

2.5.6 getch.c

The getch function gets a character from a file, and returns the character as *ch*. The function call is **getch(fdi)**, where *fdi* is a unique identifier associated with the file.

This function is not currently used.

Called By: none

Routines Called:	cmd	
Parameters:	int	fdi
Returns:	ch	

2.5.7 overlay_setup.c

The `overlay_setup` function sets up calibration, M1 and M2 gunner overlays, and M1 and M2 gun barrel overlays. It also generates DTP code for the overlays. This function is called when the message from the Simulation Host is `MSG_OVERLAY_SETUP`.

The function call is `overlay_setup(pmsg)`, where *pmsg* is a pointer to the `MSG_OVERLAY_SETUP` message.

`overlay_setup` does the following:

- Calls `make_m1_overlays` or `make_m2_overlays` (based on the type specified in the message) to create the gunner and gun barrel overlays.
- Inserts the gun barrel data into the viewport parameter nodes.

Overlays are hard-coded displays of three-dimensional polygons that are displayed on a viewport, super-imposed over the view of the terrain. The overlay shows non-terrain objects that would normally be seen when looking outside the vehicle's window. For example, gun overlays show those parts of the simulated vehicle that would be visible from the window, obscuring the view of the terrain. Gunner overlays show cross-hairs and numerical readouts of simulation parameters.

The branch mask of the configuration node to which viewport parameters are attached determines whether the corresponding viewport displays an overlay. Any node that has viewport parameters and has bit 0 of the node's branch mask set has the gunner's overlay placed on the viewport. Similarly, any node that has viewport parameters and has bit 1 of the node's branch mask set has the gun barrel added to its processing.

Gunner, gun barrel, and calibration overlays are used by T backends only. Overlays on TX backends are generated through the 2-D overlay compiler.

Called By:	cig_config	
Routines Called:	make_m1_overlays make_m2_overlays printf	
Parameters:	MSG_OVERLAY_SETUP	*pmsg
Returns:	none	

2.5.8 process_vflags.c

The process_vflags function processes system view flags (values used to turn CRT monitors on and off, and to control viewing modes such as thermal/daylight TV). process_vflags also processes the branch values indexed by the branch_index for all conditional nodes in the configuration tree; these are stored in the system view flags array. The function puts the initial view flags and branch values in the configuration tree, and updates them each frame.

Note: The process_vflags function is no longer used. The MSG_VIEW_FLAGS message is processed by msg_view_flags and various Backend Processing functions.

The function call is process_vflags(vflag, brvalues), where:

vflag is the view flags
brvalues is the branch value array

process_vflags the following:

- Gets a pointer to the root configuration node.
- Sets up the view modes for DTP.
- Loads the branch values into the view flags array.
- If the view flags have changed since the previous message/frame:
 - Loads the new view flags into the view flags array.
 - If a Force board is present, puts the video control commands in Force memory.
 - If a second backend is present, copies the new channel status to the second AAM.

Called By: none

Routines Called: AAM2_ADDR
be_query_buffer_offset
vpt_cnode_qroot

Parameters: UNS_4 vflag
UNS_4 brvalues[]

Returns: none

2.5.9 process_vpops.c

The process_vpops function sets up the position (the x, y, and z coordinates of the centroid) of the simulated vehicle in the world. This position is used to determine whether new load modules need to be read into active area memory. It is also used when preparing local terrain messages for the Simulation Host. process_vpops is called when a world/hull

matrix node (a child of the root node) is created or updated (e.g., in response to a matrix message).

The function call is **process_vppos(config_node, time_stamp, pmtx)**, where:

config_node is a pointer to the configuration node (always the world/hull node)
time_stamp is the time stamped on the message from the Simulation Host
pmtx is the node's new matrix

The simulated vehicle's position is stored in an array. This structure allows for multiple vehicles. Currently, only one simulation vehicle is supported; therefore, there is only one element in the array. The viewport positions array is pointed to by the root node's sibling pointer.

process_vppos takes the matrix provided by the Simulation Host and converts it into world coordinates. The algorithm used to do this depends on the matrix type, as follows:

RTS4x3_TYPE

Given a world-to-view matrix of:

```
| r00 r01 r02 0 |
| r10 r11 r12 0 |
| r20 r21 r22 0 |
| tx  ty  tz  1 |
```

The location of the vehicle in the world is:

```
vppos.x = -(tx,ty,tz)*(r00,r01,r02)
vppos.y = -(tx,ty,tz)*(r10,r11,r12)
vppos.z = -(tx,ty,tz)*(r20,r21,r22)
```

RTS3x3_TYPE

The location of the vehicle in the world is:

```
vppos.x = -pmtx->rts3x3.translation.x
vppos.y = -pmtx->rts3x3.translation.y
vppos.z = -pmtx->rts3x3.translation.z
```

ROT2x1_TYPE

No conversion is required.

process_vppos does the following:

- Gets a pointer to the configuration tree's root node.
- Converts the node's matrix based on its matrix type, as described above.
- Sets the global variables for the vehicle's coordinates (*my_int_x*, *my_int_y*, and *my_int_z*).
- Calls **sim_bal_tf_veh_update** to give Ballistics the new position of the simulated vehicle, for the purposes of terrain feedback reporting.

Called By: vpt_cnode_process
 vpt_update_mtx
 vpt_update_rot

Routines Called: be_query_buffer_offset

sim_bal_tf_veh_update
vpt_cnode_qroot

Parameters:

CONFIGURATION_NODE
INT_2
MTXUNION

*config_node
time_stamp
*pmtx

Returns:

none

2.6 ESIFA Interface (/cig/libsrc/libesifa)

The functions in the ESIFA Interface CSC establish and maintain a communications path between the real-time software and the ESIFA (Enhanced Subsystem Interface Adapter) board. The ESIFA, in turn, communicates with the other boards in the 9U, including the Pixel Processor Memory (PPM) and the Pixel Processor Tiler (PPT).

Each backend contains one ESIFA board. The ESIFA is a high-speed, 32-bit data path interface between the 6U Timing & Control board and the 9U backplane. The ESIFA converts the differential signals received from and sent to the 6U into the TTL-level signals used on the 9U backplane.

The ESIFA is responsible for the following:

- Returning laser range depth values to the real-time software on a T backend when laser range processing is requested. A hard-wired pixel location is used. (On TX backends, the MPV Interface functions process laser range requests from Simulation Host-specified pixel locations.)
- Downloading texture maps to the PPTs.
- Processing MSG_PPM_* messages sent by the Simulation Host to change PPM display modes, display offsets, pixel location, or pixel state.
- Processing MSG_SUBSYS_MODE messages from the Simulation Host to change sky color (fade values).
- Processing MSG_VIEWPORT_UPDATE messages from the Simulation Host to turn video channels off or on.

The structure used to maintain information on each ESIFA object is the `esifa_table` array. Each element in the array specifies that ESIFA's fade values, laser range values, port values, and a flag indicating whether or not laser range data is required this frame.

All data to be downloaded to the PPM and PPTs is first moved to ESIFA random access memory (RAM). A queue of RAM addresses waiting to be downloaded is maintained. At the end of each frame, the data is downloaded from ESIFA RAM to the subsystem boards.

Most requests that are processed through the ESIFA originate with a message from the Simulation Host. For example, the MSG_PPM_* messages are used to pass information to the PPM board. In general, data that is to be downloaded to another subsystem board is processed as follows:

1. The Message Processing function calls `esifa_queue_data` to copy the data from the Simulation Host's message into a queue (the "RAM queue").
2. The Message Processing function also calls `esifa_queue_download` to put the RAM address into another queue (the "download queue").
3. At the end of each frame, `backend_send_req` calls `esifa_send_req`.
4. `esifa_send_req` calls `esifa_send_queue`.

5. `esifa_send_queue` calls `esifa_write` to move all data in the RAM queue to the specified addresses in ESIFA RAM.
6. `esifa_send_queue` calls `esifa_download` to download the data in RAM to the appropriate 9U boards. `esifa_download` uses the addresses in the download queue to find the data in the ESIFA RAM.

The ESIFA Interface functions use Ready Systems' IFX routines to send commands to the ESIFA. (IFX provides a mechanism by which applications can interface to devices as if they were files.)

The ESIFA board has five ports that are used to store values related to viewport displays. The ports are:

- 0 video
- 1 sky
- 2 thermal
- 3 laser_set
- 4 laser_lsb or laser_msb

Figure 2-10 identifies the CSUs in the ESIFA Interface CSC. The functions performed by these CSUs are described in this section.

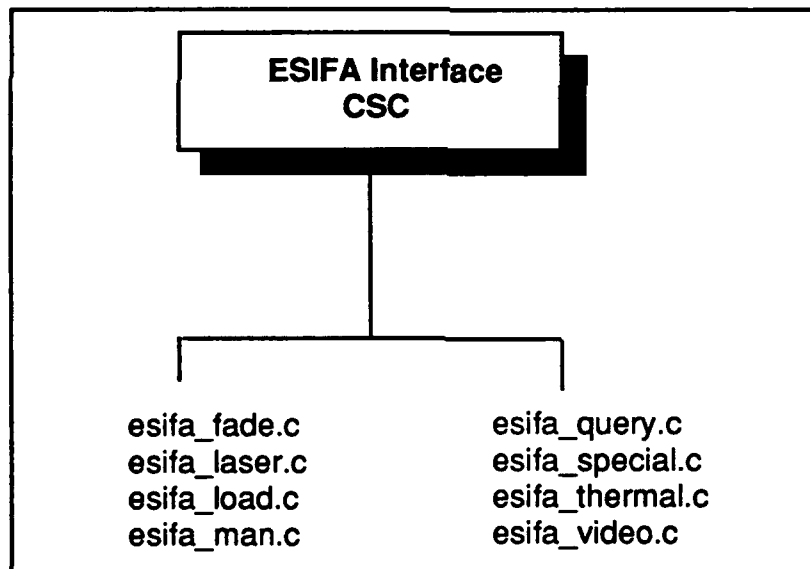


Figure 2-10. ESIFA Interface CSUs

2.6.1 `esifa_fade.c` (`esifa_set_fade`)

The `esifa_set_fade` function handles backend fade table requests. This function is called when the Simulation Host sends a `MSG_SUBSYS_MODE` message to change parameters specific to a subsystem (backend). One of the parameters specified in this message is the fade value to be displayed by all viewports in the subsystem.

The function call is `esifa_set_fade(backend, fade_and, fade_or)`, where:

backend is the backend identifier
fade_and is the `and_fade` value from the fade table
fade_or is the `or_fade` value from the fade table

`esifa_set_fade` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Sets the new fade values.

Called By: `msg_subsys_mode`

Routines Called: `esifa_get_object_addr`

Parameters:	<code>UNS_4</code>	<code>backend</code>
	<code>UNS_2</code>	<code>fade_and</code>
	<code>UNS_2</code>	<code>fade_or</code>

Returns: `none`

2.6.2 `esifa_laser.c`

The functions in the `esifa_laser.c` CSU process laser range requests from the Simulation Host for T backends. These functions are:

- `esifa_laser_request_range`
- `esifa_laser_return`

2.6.2.1 `esifa_laser_request_range`

The `esifa_laser_request_range` function sets up the ESIFA to calculate laser range depth for a specified backend/channel, to be reported to the Simulation Host every frame. This function is called if the Simulation Host sends a `MSG_LASER_REQUEST_RANGE` message for a channel on a T backend.

The function call is `esifa_laser_request_range(backend, channel)`, where:

backend is the backend id
channel is the channel number

`esifa_laser_request_range` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Sets the ESIFA variables required to trigger laser range processing.

The function returns 0 if successful. It returns EOF if the ESIFA object cannot be found.

Called By: backend_laser_request_range

Routines Called: esifa_get_object_addr

Parameters: UNS_4 backend
UNS_4 channel

Returns: 0
EOF

2.6.2.2 esifa_laser_return

The `esifa_laser_return` function returns a previously requested laser range value. This function is called at the end of each frame for every channel on a T backend for which laser request processing has been enabled.

The function call is `esifa_laser_return(backend)`, where *backend* is the backend id for which laser range data was requested.

`esifa_laser_return` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Uses the `ifx_ioctl` function to read the ESIFA ports.
- Determines the range value.
- Converts the laser depth value to floating point.

The function returns the laser depth value as *rangef* if successful. It returns -1.0 if the ESIFA board cannot be found or the range is invalid.

Called By: backend_response

Routines Called: esifa_get_object_addr
FXTO881
ifx_ioctl
printf (in debug mode only)

Parameters: UNS_4 backend

Returns: -1
rangef

2.6.3 esifa_load.c

The `esifa_load` function downloads texture files to the Pixel Processor Tilers (PPTs). This function is called if the `d` ("download textures maps to IPPTs") argument is entered on the command line used to initialize the CIG.

The function call is `esifa_load(backend, file, mask)`, where:

backend is the backend id

file is the name of the textures file to be downloaded

mask is <<TBD>>

`esifa_load` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Opens the specified file in read-only mode.
- Reads the ESIFA configuration data from the ESIFA configuration file.
- Reads the file header in the textures file to determine the file length and board type.
- If the board is a PPM, gets the PPM filename and sets `ppm_download_flag` to TRUE.
- Calls `esifa_write` to load the entire file into ESIFA RAM.
- Calls `esifa_download` to download the data from ESIFA RAM to the specified board.
- If the `ppm_download_flag` is set, calls `ppm_init` to initialize the PPM object with the specified file.
- Closes the file.

If successful, or if an error is returned from one of the file I/O routines, `esifa_load` returns the status returned from the last routine as *status*. It returns EOF if the ESIFA cannot be found. It returns -1 if it could not find the file length or board type in the file header.

Called By: `load_esifa`

Routines Called: `esifa_download`
`esifa_get_object_addr`
`esifa_write`
`ifx_close`
`ifx_ioctl`
`ifx_open`
`ifx_read`
`ppm_init`
`printf`
`search`
`strchr`
`strtol`

Parameters:	<code>UNS_4</code>	<code>backend</code>
	<code>UNS_1</code>	<code>*file</code>
	<code>INT_4</code>	<code>mask</code>

Returns: status
 EOF
 -1

2.6.4 esifa_man.c

The functions in the esifa_man.c CSU are responsible for managing communication with the ESIFAs. These functions are:

- esifa_setup
- esifa_sim_init
- esifa_send_req
- esifa_get_object_addr
- esifa_ConfigData
- esifa_queue_data
- esifa_queue_download
- esifa_send_queue

2.6.4.1 esifa_setup

The esifa_setup function establishes the ESIFA communications path associated with a specified backend. This function is called when a backend is initialized.

The function call is **esifa_setup(backend, device)**, where:

backend is the backend id
device is the device name of the ESIFA board

esifa_setup does the following:

- Allocates memory for the ESIFA table.
- Opens the specified device in read-write mode.
- Initializes the ESIFA and ports values.
- Reads the configuration data from the ESIFA into the ConfigData[] array.

The function returns 0 if successful. If an error is returned from an I/O routine, the error is returned as *status*. The function returns 1 if the device could not be opened.

Called By: backend_setup

Routines Called: free
 ifx_ioctl
 ifx_open
 malloc
 printf

Parameters: UNS_4 backend

UNS_1

*device

Returns: status
 0
 1

2.6.4.2 **esifa_sim_init**

The **esifa_sim_init** function prepares the ESIFA to run a simulation. This function is called when the backend is being readied for a simulation.

The function call is **esifa_sim_init()**. For each ESIFA, the function does the following:

- Finds the ESIFA object's entry in the **esifa_table[]** array.
- Initializes the ESIFA object's video, sky, and thermal values.
- Calls **mpvideo_get_object_addr** to determine whether the backend contains an MPV board.
 - If the returned pointer is NULL (indicating a T backend), sets the ESIFA object's laser value to 0xfff.
 - If the returned pointer is not NULL (indicating a TX backend), sets the ESIFA object's laser value to 0x2000.
- Initializes the object's **io_req** flag to TRUE.

Called By: **backend_sim_init**

Routines Called: **mpvideo_get_object_addr**

Parameters: none

Returns: none

2.6.4.3 **esifa_send_req**

The **esifa_send_req** function performs any requested I/O with the ESIFA boards. This function is called at the end of every frame, or if the backend is reset. It is also called during the configuration state if the Simulation Host sends a PPM message.

The function call is **esifa_send_req()**. The function does the following:

- If the ESIFA object's **io_req** flag is TRUE:
 - Sets the **io_req** flag to FALSE.
 - Sets all ESIFA port values.
- Calls **esifa_send_queue** to write the all queued data to ESIFA RAM and then download it to the appropriate board.

Called By: **backend_reset**

backend_send_req
db_mcc_setup

Routines Called: esifa_send_queue
ifx_ioctl

Parameters: none

Returns: none

2.6.4.4 esifa_get_object_addr

The `esifa_get_object_addr` function returns a pointer to the ESIFA object. The ESIFA objects are maintained in an array (`esifa_table[]`) indexed by backend id.

The function call is `esifa_get_object_addr(backend)`, where *backend* is the identifier of the backend that contains the ESIFA.

The function returns NULL if the ESIFA object cannot be found.

Called By: esifa_download
esifa_laser_request_range
esifa_laser_return
esifa_load
esifa_read
esifa_read_ports
esifa_set_fade
esifa_set_special
esifa_set_thermal
esifa_set_video
esifa_write
esifa_write_ports
ppm_load

Routines Called: none

Parameters: UNS_4 backend

Returns: esifa_table[n]
NULL

2.6.4.5 esifa_ConfigData

The `esifa_ConfigData` function returns a pointer to an ESIFA's configuration data. Configuration data is maintained in the `ConfigData[]` array, indexed by backend. This

function is called if the Gossip user selects the **e** ("esifa info") or **p** ("ppm load") option from the PPM Query menu.

The function call is **esifa_ConfigData(backend)**, where *backend* is the backend id.

The function returns NULL if the ESIFA has not yet been configured by **esifa_setup**.

Called By:	gos_ppm_query ppm_load	
Routines Called:	none	
Parameters:	UNS_4	backend
Returns:	&ConfigData[backend] NULL	

2.6.4.6 esifa_queue_data

The **esifa_queue_data** function copies data to a queue, to wait to be downloaded to the ESIFA RAM. This function is called by the Message Processing routines that process the PPM messages sent by the Simulation Host.

The function call is **esifa_queue_data(backend, source_address, byte_count, esifa_ram_address)**, where:

backend is the backend id
source_address is the address of the data to be copied to ESIFA RAM
byte_count is the number of bytes to be copied to ESIFA RAM
esifa_ram_address is a pointer to the address in ESIFA RAM the data is put into; this address is set by **esifa_queue_data**

esifa_queue_data does the following:

- Copies the data to the next available location in ESIFA RAM.
- Puts the data's RAM address into the location specified by *esifa_ram_address*.
- Increments its RAM pointer by the size of the data just added.

The function always returns 0.

Called By:	msg_ppm_display_mode msg_ppm_display_offset msg_ppm_pixel_location msg_ppm_pixel_state	
Routines Called:	bcopy printf	(in debug mode only)

Parameters:	UNS_2 UNS_4 INT_4 UNS_4	backend source_address byte_count *esifa_ram_address
-------------	----------------------------------	---

Returns: 0

2.6.4.7 esifa_queue_download

The `esifa_queue_download` function creates an entry in a queue that identifies a RAM address that contains data to be downloaded to the ESIFA. This function is called by the Message Processing routines that handle PPM messages sent by the Simulation Host. The routines first call `esifa_queue_data` to put the data into ESIFA RAM, then call `esifa_queue_download` to put the data's RAM address (returned by `esifa_queue_data`) into the download queue.

The function call is `esifa_queue_download(backend, esifa_ram_address, byte_count, board_addr)`, where:

backend is the backend id

esifa_ram_address is the data's starting address in ESIFA RAM (set by `esifa_queue_data`)

byte_count is the number of bytes of data to be downloaded

board_addr is the address of the target PPM board

The function always returns 0.

Called By:	msg_ppm_display_mode msg_ppm_display_offset msg_ppm_pixel_location msg_ppm_pixel_state
------------	---

Routines Called: none

Parameters:	INT_2 UNS_4 UNS_4 UNS_2	backend esifa_ram_address byte_count board_addr
-------------	----------------------------------	--

Returns: 0

2.6.4.8 esifa_send_queue

The `esifa_send_queue` function calls the appropriate routines to write the data from the RAM queue to ESIFA RAM, and then download the data from ESIFA RAM to the appropriate 9U board. This function is called at the end of every frame.

The function call is `esifa_send_queue()`. The function does the following:

- For each backend, calls `esifa_write` to write all data in the RAM queue to ESIFA RAM.
- For each index in the download queue, calls `esifa_download` to download the data from ESIFA RAM to the appropriate board.

The function returns the status returned from the last called routine as *status*.

Called By: `esifa_send_req`

Routines Called: `esifa_download`
`esifa_write`

Parameters: `none`

Returns: `status`

2.6.5 esifa_query.c

The functions in the `esifa_query.c` CSU read data from and write data to the ESIFAs. These functions are:

- `esifa_read_ports`
- `esifa_write_ports`
- `esifa_read`
- `esifa_write`
- `esifa_download`

2.6.5.1 esifa_read_ports

The `esifa_read_ports` function reads the current values set in the ESIFA ports into specified locations. This function is called if the Gossip user selects the 1 ("read Esifa io ports") option from the GT Hardware menu.

The function call is `esifa_read_ports(backend, port0, port1, port2, port3, port4)`, where:

backend is the backend id

port0 is a pointer to the location for the video port value

port1 is a pointer to the location for the sky port value
port2 is a pointer to the location for the thermal port value
port3 is a pointer to the location for the laser_set port value
port4 is a pointer to the location for the laser_lsb or laser_msb port value

esifa_read_ports does the following:

- Calls *esifa_get_object_addr* to get a pointer to the ESIFA object.
- Calls *ifx_ioctl* to read the port values.
- Puts the port values into the locations specified in the call. *gos_system* then displays the values to the Gossip user.

The function returns the value returned from *ifx_ioctl* as *status*. It returns EOF if the ESIFA cannot be found.

Called By: *gos_system*

Routines Called: *esifa_get_object_addr*
ifx_ioctl

Parameters:	UNS_4	backend
	UNS_2	*port0
	UNS_2	*port1
	UNS_2	*port2
	UNS_2	*port3
	UNS_2	*port4

Returns: *status*
 EOF

2.6.5.2 *esifa_write_ports*

The *esifa_write_ports* function writes specified values to the ESIFA ports. This function is called if the Gossip user selects the 2 ("write Esifa io ports") option from the GT Hardware menu. The user is prompted for the new values.

The function call is *esifa_write_ports(backend, port0, port1, port2, port3)*, where:

backend is the backend id
port0 is the new value for the video port
port1 is the new value for the sky port
port2 is the new value for the thermal port
port3 is the new value for the laser_set port

esifa_write_ports does the following:

- Calls *esifa_get_object_addr* to get a pointer to the ESIFA object.
- Calls *ifx_ioctl* to write the specified values to the ESIFA ports.

The function returns the value returned from `ifx_ioctl` as *status*. It returns EOF if the ESIFA cannot be found.

Called By:	gos_system		
Routines Called:	esifa_get_object_addr ifx_ioctl		
Parameters:	UNS_4	backend	
	UNS_2	port0	
	UNS_2	port1	
	UNS_2	port2	
	UNS_2	port3	
Returns:	status		
	EOF		

2.6.5.3 esifa_read

The `esifa_read` function reads data from a specified address in ESIFA RAM into a specified location. This function is called if the Gossip user selects the `r` ("read ESIFA ram") option from the PPM Query menu.

The function call is `esifa_read(backend, esifa_ram_address, byte_count, dest_addr)`, where:

backend is the backend id
esifa_ram_address is the starting address to read
byte_count is the number of bytes to read
dest_addr is the location to place the data read from the ESIFA

`esifa_read` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Calls `ifx_ioctl` to read the specified number of bytes from the ESIFA RAM into the location specified in the call.

The function returns the value returned from `ifx_ioctl` as *status*. It returns EOF if the ESIFA cannot be found.

Called By:	gos_ppm_query		
Routines Called:	esifa_get_object_addr ifx_ioctl printf		

Parameters:	UNS_4 UNS_4 UNS_4 UNS_1	backend esifa_ram_address byte_count *dest_addr
Returns:	status EOF	

2.6.5.4 esifa_write

The `esifa_write` function writes data to ESIFA RAM. This function is called at the end of each frame to write all data queued for the ESIFA during that frame from the RAM queue to RAM. It is also called if the Gossip user selects the w ("write ESIFA ram") option from the PPM Query menu.

The function call is `esifa_write(backend, esifa_ram_address, byte_count, src_addr)`, where:

backend is the backend id
esifa_ram_address is the destination address in ESIFA RAM
byte_count is the number of bytes to be written
src_addr is the source location of the data to be written

`esifa_write` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Calls `ifx_ioctl` to write the specified number of bytes to the specified location in ESIFA RAM.

The function returns the value returned from `ifx_ioctl` as *status*. It returns EOF if the ESIFA cannot be found.

Called By:	esifa_load esifa_send_queue gos_ppm_query ppm_load	
Routines Called:	esifa_get_object_addr ifx_ioctl printf	
Parameters:	UNS_4 UNS_4 UNS_4 UNS_1	backend esifa_ram_address byte_count *src_addr
Returns:	EOF	

status

2.6.5.5 esifa_download

The `esifa_download` function downloads data from ESIFA RAM to the various boards (e.g., PPM and PPT) in the 9U. This function is called at the end of each frame, to download all data that accumulated during that frame. (`esifa_download` is called after `esifa_write` is called to move the data into ESIFA RAM.) This function is also called if the Gossip user selects the `d` ("Download parameters") option from the PPM Query menu.

The function call is `esifa_download(backend, esifa_ram_addr, byte_count, backend_board_addr, flags)`, where:

backend is the backend id

esifa_ram_addr is the starting location of the data in ESIFA RAM

byte_count is the number of bytes of data to be downloaded

backend_board_addr is the address of the 9U board that is to receive the data

flags is <<TBD>>

`esifa_download` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Calls `ifx_ioctl` to download the specified number of bytes to the specified subsystem board.

The function returns the value returned from `ifx_ioctl` as *status*. It returns EOF if the ESIFA cannot be found.

Called By:

`esifa_load`
`esifa_send_queue`
`gos_ppm_query`
`ppm_load`

Routines Called:

`esifa_get_object_addr`
`ifx_ioctl`
`printf`

Parameters:

<code>UNS_4</code>	<code>backend</code>
<code>UNS_4</code>	<code>esifa_ram_addr</code>
<code>UNS_4</code>	<code>byte_count</code>
<code>UNS_4</code>	<code>backend_board_addr</code>
<code>UNS_4</code>	<code>flags</code>

Returns:

`EOF`
`status`

2.6.6 esifa_special.c (esifa_set_special)

The `esifa_set_special` function handles special viewport changes. This function is called if the Simulation Host sends a `MSG_SUBSYS_MODE`.

The function call is `esifa_set_special(backend, sky_and, sky_or, laser_and, laser_or)`, where:

backend is the backend id
sky_and is the new sky_and fade value
sky_or is the new sky_or fade value
laser_and is the new laser_and fade value
laser_or is the new laser_or fade value

`esifa_set_special` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Sets the new values.
- Sets the ESIFA's `io_req` flag to TRUE.

Called By: `backend_set_color`
 `msg_subsys_mode`

Routines Called: `esifa_get_object_addr`

Parameters:	<code>UNS_4</code>	<code>backend</code>
	<code>UNS_2</code>	<code>sky_and</code>
	<code>UNS_2</code>	<code>sky_or</code>
	<code>UNS_2</code>	<code>laser_and</code>
	<code>UNS_2</code>	<code>laser_or</code>

Returns: none

2.6.7 esifa_thermal.c (esifa_set_thermal)

The `esifa_set_thermal` function handles thermal changes. This function is called when the Simulation Host sends a `MSG_VIEWPORT_UPDATE` message to change viewport modifier information (thermal white hot, thermal black hot, etc.).

The function call is `esifa_set_thermal(backend, channel, thermal_flag, white_hot_flag)`, where:

backend is the backend id
channel is the channel number
thermal_flag is 0 (alternate mode is disabled) or 1 (alternate mode is enabled)
white_hot_flag is 0 (modifier to alternate is off) or 1 (modifier to alternate is on)

`esifa_set_thermal` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Turns the channel's thermal view ON or OFF, as indicated by the *thermal_flag*.
- Sets *white_hot* or *black_hot* mode.
- Sets the ESIFA's *io_req* flag to TRUE.

Called By: `backend_set_thermal`

Routines Called: `esifa_get_object_addr`

Parameters:	UNS_4	backend
	UNS_4	channel
	UNS_4	thermal_flag
	UNS_4	white_hot_flag

Returns: `none`

2.6.8 `esifa_video.c` (`esifa_set_video`)

The `esifa_set_video` function turns the video monitors off and on. This function is called to turn the videos off at the end of a simulation. It is also called if the Simulation Host sends a `MSG_VIEWPORT_UPDATE` message to change viewport modifier information during a simulation.

The function call is `esifa_set_video(backend, channel, flag)`, where:

backend is the backend id
channel is the channel number
flag is 0 (turn video off) or 1 (turn video on)

`esifa_set_video` does the following:

- Calls `esifa_get_object_addr` to get a pointer to the ESIFA object.
- Sets the specified channel's video off or on as specified by *flag*.
- Sets the ESIFA's *io_req* to TRUE.

Called By: `backend_reset`
`backend_set_video`

Routines Called: `esifa_get_object_addr`

Parameters:	UNS_4	backend
	UNS_4	channel
	UNS_4	flag

Returns: none

2.7 Stand-Alone Host Emulator (/cig/libsrc/libflea)

Flea is an embedded, stand-alone, Simulation Host emulator that resides within the CIG real-time software. Flea permits the CIG user to generate visual images, execute specific features, and test limited functionality without interacting with a Simulation Host. Flea is available only in stand-alone operation mode (i.e., when the system is not being driven through a simulation).

The basic startup sequence for Flea is as follows:

1. On the startup command line, the user includes the "f" argument to specify Flea mode.
2. The Task Initialization CSC creates and starts flea and flea_io_task (in addition to gossip and the real-time processing tasks). It also calls a Host Interface function to initialize the CIG-Flea interface.
3. The flea task initializes various parameters, then suspends itself. flea_io_task also suspends itself.
4. Using a VT100-compatible terminal connected to the CIG, the user accesses the Gossip main menu and selects the f ("enter Flea menu") option.
5. The gossip_tick function calls various Flea functions to invoke the Flea user interface, wake up the suspended flea task, and display the "Flea>" user prompt.

Once Flea is running, it looks for a configuration file to build the viewport configuration tree and set up the simulation environment. The configuration file is an ASCII file that contains commands in a predefined format. Various Flea routines are called to convert the information in the file to Simulation Host-type messages which are sent to the real-time software for processing.

After the configuration file has been processed, flea generates a CIG Control message to transition the CIG into the simulation state.

Once in the simulation state, the Flea user enters commands through the keyboard to move the simulated vehicle over the terrain. The user can also enter commands to change various simulation parameters (modify viewports, add or delete static vehicles, fire rounds, request terrain feedback data, etc.).

The commands entered by the Flea user during the exercise generate messages that are sent to the real-time software and processed as if they were sent by the Simulation Host. For details on the purpose and content of each message, refer to the "GT100 CIG to Simulation Host Interface Manual." Flea exchanges message packets with the real-time software (using the Host Interface routines) every frame.

The structures used to transfer message packets are the following:

flea_imsg

The packet sent from Flea to the real-time software (SIM-to-CIG messages).

flea_omsg

The packet sent from the real-time software to Flea (CIG-to-SIM messages).

The work buffers used to process messages are **p_flea_in** (for messages sent to Flea) and **p_flea_out** (for messages to be sent by Flea). The **p_flea_out** structure contains flags used to indicate that a certain type of message is to be generated, based on input from the user.

During a Flea exercise, the user selects the desired commands from a series of menus. A separate function processes the commands entered on each menu. The function responsible for a particular menu is called using the ***flea_menu** function pointer. The command prompt at the bottom of the Flea console screen identifies the menu/process that is currently active.

In addition to letting the user drive the simulated vehicle via the keyboard, Flea supports a demonstration mode and a script playback mode.

Figure 2-11 identifies the CSUs in the Flea CSC. These CSUs are described in this section.

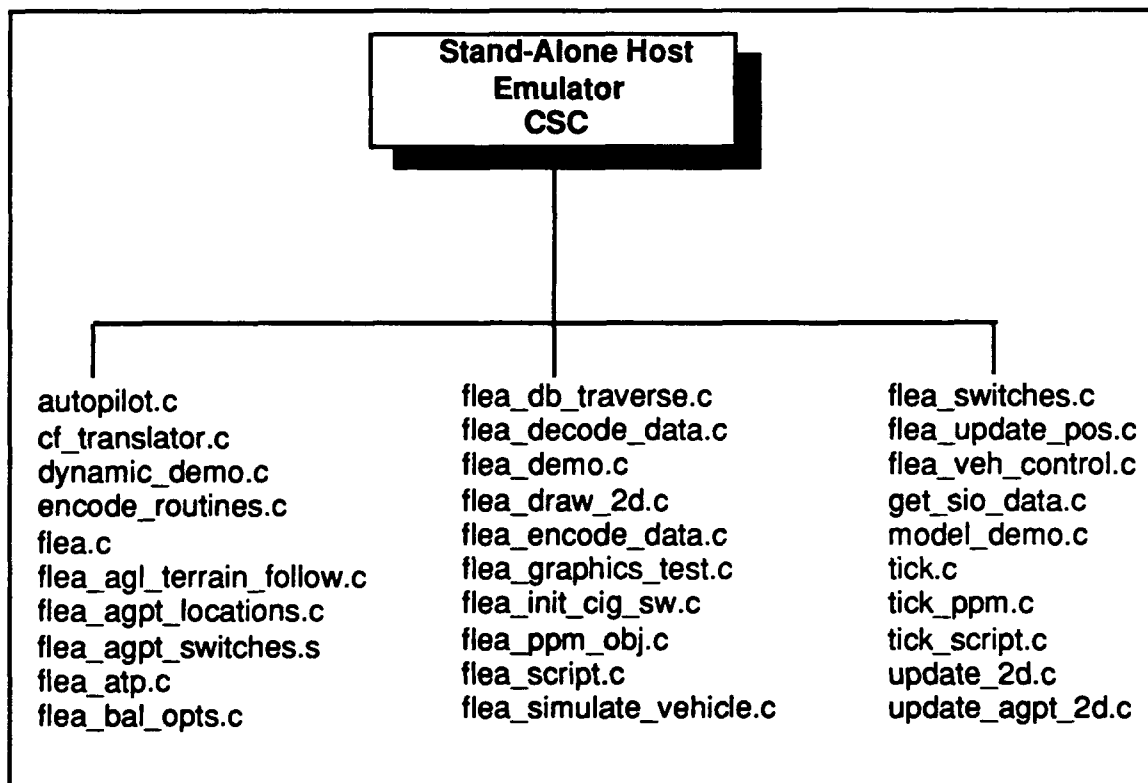


Figure 2-11. Flea CSUs

2.7.1 autopilot.c

The autopilot function demonstrates a simulated vehicle taking off, flying, maneuvering, and eventually landing. This function is called at the end of every frame if a demonstration has been initiated (demonstration = 1).

The autopilot demonstration takes the simulated vehicle through a series of actions. Each action is performed for a set of time called a segment. The full demo is divided into 25 segments of varying lengths. The length of each segment is based on the frame rate. For example, segment 0 is four times the frame rate, or four frames. Segment 1 is the next 10 frames, segment 2 is the next 15, and so on. The duration of each segment is specified by the autopilot function.

During each segment, the simulated vehicle is moved and/or rotated in a particular direction. In segment 1, for example, the vehicle turns and moves down the runway. In segment 2, the vehicle accelerates down the runway. In segment 3, the vehicle rotates up to gain altitude. The demo continues through a series of actions, ending with the vehicle landing and returning to the hangar.

The function call is `autopilot(frame)`, where *frame* is the simulation frame number. `autopilot` does the following:

- If the frame number is 0 (indicating the start of a new demo):
 - Sets up initial conditions (frame rate, vehicle position, etc.).
 - Allocates memory for the autopilot data.
- If the frame number is from 1 through 9998 (i.e., during the demo):
 - If the frame rate has changed, sets the rate and initializes the autopilot segment data; sets the deceleration rate and trim speed based on the frame rate; sets the acceleration rate.
 - Processes each segment.
- If the frame number is 9999 (indicating the end of the demo):
 - Deallocates the memory allocated for the autopilot data.

The function returns the last frame number in segment 25 — this is the total number of frames the demo will run. `flea_demo` uses this value to determine when to stop the demo. The function returns -1 if not enough memory was available for the autopilot information.

Called By: `flea_demo`

Routines Called: `calloc`
`cos`
`free`
`GLOB`
`printf`
`sin`
`TORAD`

Parameters: `INT_4` `frame`

Returns: pdemo->segment[25]
-1

2.7.2 cf_translator.c

The functions in the cf_translator.c CSU are used to read and process an ASCII file that contains CIG configuration messages. A Simulation Host-type message is generated for each entry in the file. These messages are placed into a temporary message buffer, and are then copied (by flea_init_cig_sw) to the current outgoing message packet. The packet is then sent to the real-time software and processed as if it were sent by the Simulation Host. This process lets the Flea user build the viewport configuration tree, set up 2-D overlays, and configure simulation parameters.

The configuration messages generated by the cf_translator functions have the same formats as the messages sent from the Simulation Host. Refer to the "GT100 CIG to Simulation Host Interface Manual" for details.

The functions in this CSU are:

- config_translator
- process_cig_ctl
- process_file_description
- process_configtree_node
- process_viewport_state
- process_define_tx_mode
- process_overlay_setup
- process_tf_init_hdr
- process_tf_init_pt
- process_agl_setup
- process_sio_init
- process_add_traj_table
- process_traj_entry
- process_2d_setup
- process_lt_state
- process_drll_pkt_size
- process_sio_close
- process_tf_state
- process_ammo_define
- process_ppm_display_mode
- process_ppm_display_offset
- process_ppm_pixel_location
- process_ppm_pixel_state
- read_a_keyword
- remove_white_space
- remove_comment_lines

The OUTPUT_MESSAGE macro, described in Appendix B, is also defined in this CSU. The process_* functions use OUTPUT_MESSAGE to place their messages into the message buffer used by flea_init_cig_sw.

2.7.2.1 config_translator

The `config_translator` function opens the Flea configuration file and calls the appropriate `process_*` function to process each line in the file. The `process_*` functions generate the Simulation Host-type messages used to configure the system. `config_translator` is called whenever Flea is invoked by the user.

The function call is `config_translator (input_fn, msg_buffer, msg_buffer_size)`, where:

input_fn is the name of the configuration file
msg_buffer is a pointer to the buffer to be used for outgoing messages
msg_buffer_size is the size of the outgoing message buffer

`config_translator` does the following:

- Opens the specified file.
- Zeroes out the command buffer.
- Calls `read_a_keyword` to read an entry from the file.
- Calls the appropriate `process_*` function to process the line.
- Verifies that the outgoing buffer has room to hold the new outgoing messages. If it does not, exits with a 1.
- Repeats the procedure for each line in the file.
- When the END command is detected in the file, or if an error is reported by one of the called procedures:
 - Closes the file.
 - Outputs "PASSED" or "FAILED" to stdout, depending on whether or not an error occurred.
 - Uses the `OUTPUT_MESSAGE` macro to put a `MSG_END` message into the message buffer.

The function returns the status from the last called routine as *error_flag*. It returns EOF if the specified file could not be opened or contained a syntax error.

Called By: `flea_init_cig_sw`

Routines Called: `bzero`
 `exit`
 `fclose`
 `fopen`
 `OUTPUT_MESSAGE`
 `printf`
 `process_2d_setup`
 `process_add_traj_table`
 `process_agl_setup`
 `process_ammo_define`
 `process_cig_ctl`
 `process_configtree_node`
 `process_define_tx_mode`
 `process_dr11_pkt_size`

```

process_file_description
process_lt_state
process_overlay_setup
process_ppm_display_mode
process_ppm_display_offset
process_ppm_pixel_location
process_ppm_pixel_state
process_sio_close
process_sio_init
process_tf_init_hdr
process_tf_init_pt
process_tf_state
process_traj_entry
process_viewport_state
read_a_keyword
strcmp
strcpy

```

Parameters:	char UNS_1 INT_4	*input_fn *msg_buffer msg_buffer_size
Returns:	EOF error_flag	

2.7.2.2 process_cig_ctl

The `process_cig_ctl` function generates the MSG_CIG_CTL message. This function is called if the entry in the input configuration file is START_CIG_CTL.

The function call is `process_cig_ctl()`. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (`control_code`, `subsys_id`, or `subsys_channel`).
 - Reads the data specified for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_CIG_CTL message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_CIG_CTL message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
 fgets
 fscanf
 OUTPUT_MESSAGE
 printf
 read_a_keyword
 strcmp

Parameters: none

Returns: error_flag

2.7.2.3 process_file_description

The process_file_description function generates the MSG_FILE_DESCR message. This function is called if the entry in the input configuration file is START_FILE_DESCR.

The function call is **process_file_description()**. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls read_a_keyword to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (size, number, request, or filename).
 - Reads the data specified for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_FILE_DESCR message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_FILE_DESCR message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
 fgets
 fscanf
 OUTPUT_MESSAGE
 printf
 read_a_keyword
 strcmp

Parameters: none

Returns: error_flag

2.7.2.4 process_configtree_node

The process_configtree_node function generates the MSG_CREATE_CONFIGNODE message. This function is called if the entry in the input configuration file is START_CREATE_CONFIGNODE or START_CONFIGTREE_NODE.

The function call is process_configtree_node(). The function does the following:

- Initializes the output message.
- Reads the node_index from the input file.
- Zeroes out the command buffer.
- Calls read_a_keyword to read each parameter from the node's entry .
- For each parameter in the entry:
 - Determines which parameter is being set (parent_index, node_type, matrix, etc.).
 - Reads the data specified for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_CREATE_CONFIGNODE message, an END_CONFIGTREE_NODE message, or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_CREATE_CONFIGNODE message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fgetc
fgets
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.5 process_viewport_state

The process_viewport_state function generates the MSG_VIEWPORT_STATE message. This function is called if the entry in the input configuration file is START_VIEWPORT_STATE.

The function call is **process_viewport_state()**. The function does the following:

- Initializes the output message.
- Reads the `node_index` from the input file.
- If the `node_index` is too large, outputs an error and sets `error_flag` to TRUE.
- Zeroes out the command buffer.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (`viewport_id`, `resolution`, `viewing_range`, etc.).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an `END_VIEWPORT_STATE` message or an invalid parameter.
- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_VIEWPORT_STATE` message into the message buffer.

The function returns *error_flag* set to FALSE if successful, TRUE if the `node_index` is invalid, or EOF if a syntax error was detected.

Called By: `config_translator`

Routines Called: `bzero`
 `fgets`
 `fscanf`
 `OUTPUT_MESSAGE`
 `printf`
 `read_a_keyword`
 `strcmp`

Parameters: `none`

Returns: `error_flag`

2.7.2.6 **process_define_tx_mode**

The `process_define_tx_mode` function generates the `MSG_DEFINE_TX_MODE` message. This function is called if the entry in the input configuration file is `START_DEFINE_TX_MODE`.

The function call is **process_define_tx_mode()**. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (`subsys_id`, `mode_id`, `mpv_mode`, etc.).
 - Reads the data for the parameter.

- Sets the parameter's value in the output message.
- Stops reading when it detects an END_DEFINE_TX_MODE message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_DEFINE_TX_MODE message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fgets
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.7 process_overlay_setup

The process_overlay_setup function generates the MSG_OVERLAY_SETUP message. This function is called if the entry in the input configuration file is START_OVERLAY_SETUP.

The function call is **process_overlay_setup()**. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls read_a_keyword to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (type, barrel_offset, etc.).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_OVERLAY_SETUP message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_OVERLAY_SETUP message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
 fgets
 fscanf
 OUTPUT_MESSAGE
 printf
 read_a_keyword
 strcmp

Parameters: none

Returns: error_flag

2.7.2.8 process_tf_init_hdr

The process_tf_init_hdr function generates the MSG_TF_INIT_HDR message. This function is called if the entry in the input configuration file is START_TF_INIT_HDR.

The function call is **process_tf_init_hdr()**. The function does the following:

- Initializes the output message.
- Reads the vehicle id from the file.
- Calls read_a_keyword to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (own_vehicle_flag, point_count, etc.).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_TF_INIT_HDR message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_TF_INIT_HDR message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
 fgets
 fscanf
 OUTPUT_MESSAGE
 printf
 read_a_keyword
 strcmp

Parameters: none

Returns: `error_flag`

2.7.2.9 `process_tf_init_pt`

The `process_tf_init_pt` function generates the `MSG_TF_INIT_PT` message. This function is called if the entry in the input configuration file is `START_TF_INIT_PT`.

The function call is `process_tf_init_pt()`. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Reads the vehicle id from the file.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (point or position).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an `END_TF_INIT_PT` message or an invalid parameter.
- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_TF_INIT_PT` message into the message buffer.

The function returns `error_flag` set to `FALSE` if successful, or `EOF` if a syntax error was detected.

Called By: `config_translator`

Routines Called: `bzero`
 `fgets`
 `fscanf`
 `OUTPUT_MESSAGE`
 `printf`
 `read_a_keyword`
 `strcmp`

Parameters: `none`

Returns: `error_flag`

2.7.2.10 `process_agl_setup`

The `process_agl_setup` function generates the `MSG_AGL_SETUP` message. This function is called if the message in the input configuration file is `START_AGL_SETUP`.

The function call is `process_agl_setup()`. The function does the following:

- Initializes the output message.

- Zeroes out the command buffer.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (for this message, the only valid parameter is state).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an `END_AGL_SETUP` message or an invalid parameter.
- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_AGL_SETUP` message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: `config_translator`

Routines Called: `bzero`
 `fgets`
 `fscanf`
 `OUTPUT_MESSAGE`
 `printf`
 `read_a_keyword`
 `strcmp`

Parameters: none

Returns: `error_flag`

2.7.2.11 `process_sio_init`

The `process_sio_init` function generates the `MSG_SIO_INIT` message. This function is called if the entry in the input configuration file is `START_SIO_INIT`.

The function call is `process_sio_init ()`. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (`device_name`, `device_id`, etc.).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an `END_SIO_INIT` message or an invalid parameter.
- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_SIO_INIT` message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.12 process_add_traj_table

The process_add_traj_table function generates the MSG_ADD_TRAJ_TABLE message. This function is called if the message in the input configuration file is START_ADD_TRAJ_TABLE.

The function call is **process_add_traj_table()**. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls read_a_keyword to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (traj_type, sample_rate, etc.).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_ADD_TRAJ_TABLE message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_ADD_TRAJ_TABLE message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.13 process_traj_entry

The process_traj_entry function generates the MSG_TRAJ_ENTRY message. This function is called if the entry in the input configuration file is START_TRAJ_ENTRY.

The function call is process_traj_entry(). The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls read_a_keyword to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (traj_type, entry_index, etc.).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_TRAJ_ENTRY message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_TRAJ_ENTRY message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.14 process_2d_setup

The process_2d_setup function generates the MSG_2D_SETUP message. This function is called if the entry in the input configuration file is START_2D_SETUP.

The function call is process_2d_setup(). The function does the following:

- Initializes the output message.

- Zeroes out the command buffer.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (for this message, the only valid parameter is string).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading the entry when it detects the `END_2D_SETUP` message.
- Stops reading when it detects an `END_2D_SETUP` message or an invalid parameter.
- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_2D_SETUP` message into the message buffer.

The function returns *error_flag* set to `FALSE` if successful, or `EOF` if a syntax error was detected.

Called By: `config_translator`

Routines Called: `bzero`
`fgetc`
`OUTPUT_MESSAGE`
`printf`
`read_a_keyword`
`strcmp`

Parameters: `none`

Returns: `error_flag`

2.7.2.15 `process_lt_state`

The `process_lt_state` function generates the `MSG_LT_STATE` message. This function is called if the entry in the input configuration file is `START_LT_STATE`.

The function call is `process_lt_state()`. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (code, size, or interval).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an `END_LT_STATE` message or an invalid parameter.
- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_LT_STATE` message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.16 process_dr11_pkt_size

The process_dr11_pkt_size function generates the MSG_DR11_PKT_SIZE message. This function is called if the entry in the input configuration file is START_DR11_PKT_SIZE.

The function call is **process_dr11_pkt_size()**. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls read_a_keyword to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (sim_to_cig_pkt_size, cig_to_sim_pkt_size, etc.).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_DR11_PKT_SIZE message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_DR11_PKT_SIZE message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword

strcmp

Parameters: none

Returns: error_flag

2.7.2.17 process_sio_close

The process_sio_close function generates the MSG_SIO_CLOSE message. This function is called if the entry in the input configuration file is START_SIO_CLOSE.

The function call is **process_sio_close()**. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls read_a_keyword to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set; for this message type, the only valid parameter is device_name.
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_SIO_CLOSE message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_SIO_CLOSE message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.18 process_tf_state

The process_tf_state function generates the MSG_TF_STATE message. This function is called if the entry in the input configuration file is START_TF_STATE.

The function call is **process_tf_state()**. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls **read_a_keyword** to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (**vehicle_id**, **code**, or **frequency**).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an **END_TF_STATE** message or an invalid parameter.
- Uses the **OUTPUT_MESSAGE** macro to put the **MSG_TF_STATE** message into the message buffer.

The function returns *error_flag* set to **FALSE** if successful, or **EOF** if a syntax error was detected.

Called By: **config_translator**

Routines Called: **bzero**
 fscanf
 OUTPUT_MESSAGE
 printf
 read_a_keyword
 strcmp

Parameters: **none**

Returns: **error_flag**

2.7.2.19 **process_ammo_define**

The **process_ammo_define** function generates the **MSG_AMMO_DEFINE** message. This function is called if the message in the input configuration file is **START_AMMO_DEFINE**.

The function call is **process_ammo_define()**. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls **read_a_keyword** to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set; for this message type, the only valid parameter is **ammo_type_map**.
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an **END_AMMO_DEFINE** message or an invalid parameter.

- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_AMMO_DEFINE` message into the message buffer.

The function returns *error_flag* set to `FALSE` if successful, or `EOF` if a syntax error was detected.

Called By: `config_translator`

Routines Called: `bzero`
`fscanf`
`OUTPUT_MESSAGE`
`printf`
`read_a_keyword`
`strcmp`

Parameters: `none`

Returns: `error_flag`

2.7.2.20 `process_ppm_display_mode`

The `process_ppm_display_mode` function generates the `MSG_PPM_DISPLAY_MODE` message. This function is called if the entry in the input configuration file is `START_PPM_DISPLAY_MODE`.

The function call is `process_ppm_display_mode()`. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (subsystem, channel, or `display_mode`).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an `END_PPM_DISPLAY_MODE` message or an invalid parameter.
- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_PPM_DISPLAY_MODE` message into the message buffer.

The function returns *error_flag* set to `FALSE` if successful, or `EOF` if a syntax error was detected.

Called By: `config_translator`

Routines Called: `bzero`
`fscanf`

OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.21 process_ppm_display_offset

The process_ppm_display_offset function generates the MSG_PPM_DISPLAY_OFFSET message. This function is called if the entry in the input configuration file is START_PPM_DISPLAY_OFFSET.

The function call is process_ppm_display_offset(). The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls read_a_keyword to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (subsystem, channel, offset_i, or offset_j).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an END_PPM_DISPLAY_OFFSET message or an invalid parameter.
- Uses the OUTPUT_MESSAGE macro to put the MSG_PPM_DISPLAY_OFFSET message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.22 process_ppm_pixel_location

The `process_ppm_pixel_location` function generates the `MSG_PPM_PIXEL_LOCATION` message. This function is called if the entry in the input configuration file is `START_PPM_PIXEL_LOCATION`.

The function call is `process_ppm_pixel_location()`. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.
- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (subsystem, channel, location_i, or location_j).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an `END_PPM_PIXEL_LOCATION` message or an invalid parameter.
- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_PPM_PIXEL_LOCATION` message into the message buffer.

The function returns `error_flag` set to `FALSE` if successful, or `EOF` if a syntax error was detected.

Called By: config_translator

Routines Called: bzero
fscanf
OUTPUT_MESSAGE
printf
read_a_keyword
strcmp

Parameters: none

Returns: error_flag

2.7.2.23 process_ppm_pixel_state

The `process_ppm_pixel_state` function generates the `MSG_PPM_PIXEL_STATE` message. This function is called if the entry in the input configuration file is `START_PPM_PIXEL_STATE`.

The function call is `process_ppm_pixel_state()`. The function does the following:

- Initializes the output message.
- Zeroes out the command buffer.

- Calls `read_a_keyword` to read each parameter from the entry.
- For each parameter in the entry:
 - Determines which parameter is being set (subsystem, channel, or state).
 - Reads the data for the parameter.
 - Sets the parameter's value in the output message.
- Stops reading when it detects an `END_PPM_PIXEL_STATE` message or an invalid parameter.
- Uses the `OUTPUT_MESSAGE` macro to put the `MSG_PPM_PIXEL_STATE` message into the message buffer.

The function returns *error_flag* set to FALSE if successful, or EOF if a syntax error was detected.

Called By: `config_translator`

Routines Called: `bzero`
`fscanf`
`OUTPUT_MESSAGE`
`printf`
`read_a_keyword`
`strcmp`

Parameters: `none`

Returns: `error_flag`

2.7.2.24 `read_a_keyword`

The `read_a_keyword` function puts the next entry in the configuration file into the command buffer. `config_translator` then parses the entry in the command buffer and calls the appropriate `process_*` function to handle it.

The function call is `read_a_keyword()`.

The function first calls `remove_comment_lines` and `remove_white_space` to eliminate extraneous data. It then reads each character into the command buffer until it encounters a pound sign (#) or a space.

Called By: `config_translator`
`process_2d_setup`
`process_add_traj_table`
`process_agl_setup`
`process_ammo_define`
`process_cig_ctl`
`process_configtree_node`
`process_define_tx_mode`
`process_dr11_pkt_size`
`process_file_description`

process_lt_state
process_overlay_setup
process_ppm_display_mode
process_ppm_display_offset
process_ppm_pixel_location
process_ppm_pixel_state
process_sio_close
process_sio_init
process_tf_init_hdr
process_tf_init_pt
process_tf_state
process_traj_entry
process_viewport_state

Routines Called: fgetc
fscanf
isspace
remove_comment_lines
remove_white_space
ungetc

Parameters: none

Returns: none

2.7.2.25 remove_white_space

The remove_white_space function checks for and skips over blank lines in the configuration file.

The function call is **remove_white_space()**.

Called By: read_a_keyword

Routines Called: fgetc
isspace
ungetc

Parameters: none

Returns: none

2.7.2.26 `remove_comment_lines`

The `remove_comment_lines` function checks for and skips over comment lines (identified by #) in the configuration file.

The function call is `remove_comment_lines()`.

Called By: `read_a_keyword`

Routines Called: `fgetc`
`fgets`
`ungetc`

Parameters: `none`

Returns: `none`

2.7.3 `dynamic_demo.c`

The `dynamic_demo` function provides a real-time demonstration of up to 32 dynamic models. This demonstration is controlled as follows:

- The demo is started if the user selects the `d` ("moving model demo") option on the Flea Switches menu. (The demo can also be invoked through the Flea AGPT Locations and Flea AGPT Switches menus.)
- If a demo has been started, `flea_encode_data` calls `dynamic_demo` each frame to continue the demo.
- The number of vehicles in the demo, and other demo parameters, can be set on the Moving Models Demonstration menu, reached by selecting the `d` ("moving model demo") option on the Flea Switches menu after a demo has been started.
- The demo can be stopped by selecting option `9` ("stop demonstration") on the Moving Models Demonstration menu.

The function call is `dynamic_demo(frame)`, where *frame* is the simulation frame number. `dynamic_demo` does the following:

- If the frame number is 0 (indicating the start of a new demo):
 - Allocates memory for the demo.
 - Initializes parameters that can be changed by the operator.
 - Sets the demonstration variable to 2.
- If the frame number is from 1 through 9998 (i.e., during a demo):
 - Updates the position and orientation of each model.
 - Builds a `MSG_OTHERVEH_STATE` for each model and puts it in the Flea-to-CIG message buffer.
- If the frame number is greater than or equal to 9999 (indicating the end of the demo):
 - Frees the allocated memory.

The function returns 1 if *frame* is 0, and returns 9 for any other *frame* value.

Called By:	flea_agpt_locations flea_agpt_switches flea_encode_data flea_switches model_demo
Routines Called:	calloc cos free printf sin TORAD
Parameters:	INT_4 frame
Returns:	1 9

2.7.4 encode_routines.c

The functions in the encode_routines.c CSU build the Simulation Host-type message packets sent to the simulation software during a Flea exercise. The messages are then processed by the real-time software as if they had been sent by the Simulation Host during an actual simulation.

Most of the routines in this CSU are called every frame during a Flea exercise. The basic processing cycle is as follows:

- During the Flea exercise, the user enters commands through various Flea menus to move the simulation vehicle, add static vehicles, fire rounds, and otherwise interact with the simulation environment.
- The flea functions that manage the user interface put the new data into the p_flea_out data structure. In most cases, the functions also set a flag in p_flea_out that indicates that new data has been added. Each type of data has its own flag.
- At the end of each frame, flea_encode_data calls all of the encode (upd_*) routines.
- Each upd_* function checks to see if its flag is on, indicating that p_flea_out contains new data for it to process.
- If its flag is set, the upd_* function gets its new data from p_flea_out and puts it into a Simulation Host-type message in the current frame's outgoing message packet.
- The upd_* function resets its flag in p_flea_out, for use the next frame.

The runtime messages generated by the upd_* functions have the same formats as messages sent from the Simulation Host. Refer to the "GT100 CIG to Simulation Host Interface Manual" for details.

The functions in this CSU are:

- upd_matrix_values
- upd_rotation_values
- upd_dynamic_matrix
- upd_view_flags
- upd_round_fired
- upd_chord_fired
- upd_auto_fire
- upd_rem_static_veh
- upd_add_static_veh
- upd_send_dynamic
- upd_count_hits_per_min
- upd_view_mode
- upd_show_eff
- upd_clouds
- upd_req_agl
- upd_req_point
- upd_req_lrange
- upd_view_mag
- upd_subsys_mode
- upd_viewport_up
- upd_send_stop
- flea_error_print
- put_in_hdr
- upd_flea_vehicles
- fire_round
- process_chord
- process_round
- cancel_round
- upd_sio_write
- upd_lt_state
- send_gun_overlay
- send_ammo_define

2.7.4.1 upd_matrix_values

The upd_matrix_values function generates a MSG_HPRXYZS_MATRIX or MSG_RTS4x3_MATRIX message to update the simulated vehicle's main transformation matrix. This is the first function called at the end of every frame during a Flea exercise.

The function call is upd_matrix_values(). The function does the following:

- Resets the packet count to 0.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Builds the MSG_HPRXYZS_MATRIX or MSG_RTS4x3_MATRIX message (depending on the defined matrix type) and puts it in the packet.

The function calls `flea_error_print` if the message packet does not contain enough room for the new message.

Matrix types are initialized to an invalid type until the operator enters the main Flea menu.

Called By: `flea_encode_data`

Routines Called: `flea_error_print`

Parameters: `none`

Returns: `none`

2.7.4.2 `upd_rotation_values`

The `upd_rotation_values` function generates `MSG_ROT2x1_MATRIX`, `MSG_1ROTATION`, and `MSG_3ROTATIONS` messages to update the transformation matrices in the viewport configuration tree (to simulate moving components). This function is used if the user selects the `e` ("transform update") option on the Flea Switches menu to initiate continual movement of a node. The user enters the node index, rotation type, rotation axis, and the number of degrees to rotate each frame.

The function call is `upd_rotation_values()`. For each node selected by the user, the function does the following:

- Validates the rotation value (number of degrees of rotation); changes it if it is invalid.
- Determines the rotation type (`ROT2x1`, `1ROTATION`, or `3ROTATIONS`).
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Builds the `MSG_ROT2x1_MATRIX`, `MSG_1ROTATION`, or `MSG_3ROTATIONS` message and puts it in the packet.

The function calls `flea_error_print` if the message packet does not contain enough room for the new message.

Called By: `flea_encode_data`

Routines Called: `cos`
`flea_error_print`
`sin`

Parameters: `none`

Returns: none

2.7.4.3 upd_dynamic_matrix

The upd_dynamic_matrix function generates a MSG_HPRXYZS_MATRIX message to update the transformation matrices of the simulated vehicle.

The function call is **upd_dynamic_matrix()**. The function does the following:

- Validates the matrix type.
- Checks to see if the node's p_flea_out->xfrm_update flag is set. This flag indicates that the transformation matrix is to be changed. It is set by tick_init after the simulation vehicle's position and orientation are entered by the user at startup.
- Validates the rotation value (number of degrees of rotation); changes it if it is invalid.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Builds the MSG_HPRXYZS_MATRIX message and puts it in the packet.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: flea_error_print

Parameters: none

Returns: none

2.7.4.4 upd_view_flags

The upd_view_flags function generates a MSG_VIEW_FLAGS message to update the system view flags and configuration tree branch values during the Flea exercise. This message is sent every frame unless the exercise is being stopped. The user can update view flags using the v ("change view flags") option on the Flea Switches menu.

The function call is **upd_view_flags()**. The function does the following:

- Checks for the stop flag in p_flea_out, to see if a CIG Control-Stop message has been generated.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Builds the MSG_VIEW_FLAGS message (using the data in p_flea_out->view_flag) and puts it in the packet.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: flea_error_print

Parameters: none

Returns: none

2.7.4.5 upd_round_fired

The upd_round_fired function is used to generate MSG_CANCEL_ROUND, MSG_ROUND_FIRED, and MSG_PROCESS_ROUND messages to send to Ballistics.

The function call is **upd_round_fired()**. The function does the following:

- Makes sure shoot_flag in p_flea_out is non-zero. This flag is set to 1 if the user selects the ! ("fire round," "shoot gun," or "fire a round") option on the Flea Ballistics, Flea Switches, Vehicle Control, or Flea main menu. It is set to -1 if the user selects the * ("cancel last round") option on the Flea Ballistics menu.
- If the round's shoot_flag is -1 (cancel round):
 - Increments the packet count (size).
 - Decrements the round count.
 - Calls cancel_round to generate a MSG_CANCEL_ROUND message.
 - Puts the message in the outgoing message packet.
- If the round fired is a new round type:
 - Increments the packet count (size).
 - Calls process_round to generate a MSG_PROCESS_ROUND message.
 - Puts the message in the outgoing message packet.
 - Increments the round count.
- If the round fired is not a new round type:
 - Increments the packet count (size).
 - Calls fire_round to generate a MSG_ROUND_FIRED message.
 - Puts the message in the outgoing message packet.
 - Increments the round count.

The function calls flea_error_print if the message packet does not contain enough room for the new message. It also sets the round's shoot_flag to FALSE.

Called By: flea_encode_data

Routines Called: cancel_round
fire_round
flea_error_print
process_round

Parameters: none

Returns: none

2.7.4.6 upd_chord_fired

The upd_chord_fired function is used to generate a MSG_PROCESS_CHORD message to send to Ballistics.

The function call is **upd_chord_fired()**. The function does the following:

- Makes sure the proc_chord_flag in p_flea_out is TRUE. This flag is set to TRUE if the user selects the I ("fire laser weapon") or L ("toggle laser weapon") option on the Flea Ballistics menu.
- Increments the packet count (size).
- Calls process_chord to generate the MSG_PROCESS_CHORD message.
- Puts the message in the outgoing packet.
- Resets proc_chord_flag to FALSE.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: flea_error_print
process_chord

Parameters: none

Returns: none

2.7.4.7 upd_auto_fire

The upd_auto_fire function is used to send continuous MSG_ROUND_FIRED and MSG_PROCESS_ROUND messages to Ballistics. Up to eight rounds can be processed per frame.

The function call is **upd_auto_fire()**. The function does the following:

- Makes sure auto_fire_flag in p_flea_out is TRUE. This flag is controlled using the @ ("toggle auto fire") option on the Flea Ballistics menu.
- Computes the rounds per minute and the number of rounds left to do.
- If the round fired is a new round type:
 - Increments the packet count (size).
 - Calls process_round to generate a MSG_PROCESS_ROUND message.
 - Puts the message in the packet.
 - Increments the round count.

- If the round fired is not a new round type:
 - Increments the packet count (size).
 - Calls fire_round to generate a MSG_ROUND_FIRED message.
 - Puts the message in the packet.
 - Increments the round count.

Called By: flea_encode_data

Routines Called: fire_round
process_round

Parameters: none

Returns: none

2.7.4.8 upd_rem_static_veh

The upd_rem_static_veh function generates a MSG_STATICVEH_REM message to delete a static vehicle from the simulation environment.

The function call is upd_rem_static_veh(). The function does the following:

- Makes sure rem_staticveh_flag in p_flea_out is TRUE. This flag is set to TRUE if the user selects the d ("remove static veh") option from the Flea Ballistics menu.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Copies the data from p_flea_out to the outgoing message packet.
- Resets rem_staticveh_flag to FALSE.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: bcopy
flea_error_print

Parameters: none

Returns: none

2.7.4.9 **upd_add_static_veh**

The **upd_add_static_veh** function generates a MSG_STATICVEH_STATE message to add a new static vehicle to the simulation environment.

The function call is **upd_add_static_veh()**. The function does the following:

- Makes sure **add_staticveh_flag** in **p_flea_out** is TRUE. This flag is set to TRUE if the user selects the s ("drop static veh") option from the Flea Ballistics menu.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Copies the data from **p_flea_out** to the outgoing message packet.
- Resets **add_staticveh_flag** to FALSE.

The function calls **flea_error_print** if the message packet does not contain enough room for the new message.

Called By: **flea_encode_data**

Routines Called: **bcopy**
 flea_error_print

Parameters: **none**

Returns: **none**

2.7.4.10 **upd_send_dynamic**

The **upd_send_dynamic** function generates a MSG_OTHERVEH_STATE message to update the position of a dynamic vehicle in the simulation environment.

The function call is **upd_send_dynamic()**. For each **otherveh_state** entry in **p_flea_out**, the function does the following:

- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Copies the data from **p_flea_out** to the outgoing message packet.

otherveh_state entries in **p_flea_out** are generated when the user selects the + ("add vehicles") or V ("add vehicleS") option from the Flea Ballistics menu.

The function calls **flea_error_print** if the message packet does not contain enough room for the new message.

Called By: **flea_encode_data**

Routines Called: bcopy
 flea_error_print

Parameters: none

Returns: none

2.7.4.11 **upd_count_hits_per_min**

The `upd_count_hits_per_min` function keeps track of the number of hits (Ballistics intersections) per minute. This value is used for auto-firing.

The function call is `upd_count_hits_per_min()`. The function does the following:

- Makes sure `hit_count_flag` in `p_flea_out` is TRUE. This flag is controlled using the @ ("toggle auto fire") option on the Flea Ballistics menu.
- Increments the `frame_count` in `p_flea_in`.
- If the frame count is greater than or equal to 5 times the frame rate:
 - Sets the `hits_per_minute` in `p_flea_in` to the `hit_count` times 12.
 - Resets the `hit_count` in `p_flea_in` to 0.
 - Resets the `frame_count` in `p_flea_in` to 0.

Called By: flea_encode_data

Routines Called: none

Parameters: none

Returns: none

2.7.4.12 **upd_view_mode**

The `upd_view_mode` function is not currently implemented.

2.7.4.13 **upd_show_eff**

The `upd_show_eff` function generates a `MSG_SHOW_EFFECT` message to display special effects at the point of a round's intersection with the simulation environment.

The function call is `upd_show_eff()`. The function does the following:

- Makes sure the hit's `show_eff_flag` in `p_flea_out` is TRUE.
- Resets the hit's `show_eff_flag` to FALSE.
- Increments the packet count (size).

- Builds the message header and puts it in the packet.
- Builds the MSG_SHOW_EFFECT message and puts it in the packet.

The function calls `flea_error_print` if the message packet does not contain enough room for the new message.

Called By: `flea_encode_data`

Routines Called: `flea_error_print`

Parameters: `none`

Returns: `none`

2.7.4.14 `upd_clouds`

The `upd_clouds` function generates a MSG_CLOUD_STATE message to change the models used for the clouds around the simulated vehicle.

The function call is `upd_clouds()`. The function does the following:

- Makes sure `cloud_flag` in `p_flea_out` is TRUE. This flag is set to TRUE if the user selects the C ("clouds message") option on the Flea Switches menu.
- Resets `cloud_flag` to FALSE.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Builds the MSG_CLOUD_STATE message and puts it in the packet.

The function calls `flea_error_print` if the message packet does not contain enough room for the new message.

Called By: `flea_encode_data`

Routines Called: `flea_error_print`

Parameters: `none`

Returns: `none`

2.7.4.15 `upd_req_agl`

The `upd_req_agl` function generates a MSG_AGL_SETUP message to enable or disable AGL (above ground level) processing. If AGL processing is enabled, the real-time

software calculates the simulated vehicle's altitude each frame and returns it to the Simulation Host (or, in this case, Flea).

The function call is **upd_req_agl()**. The function does the following:

- Makes sure the req_agl flag in p_flea_out is TRUE. This flag can be set TRUE using the a ("request agl") or g ("toggle agl ground follow") option on the Flea Ballistics menu, or the a ("request agl") option on the Flea Switches menu. It can be set FALSE using the A ("disable agl") option on the Flea Ballistics menu.
- Resets the req_agl flag to FALSE.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Builds the MSG_AGL_SETUP message and puts it in the packet.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: flea_error_print

Parameters: none

Returns: none

2.7.4.16 upd_req_point

The upd_req_point function generates a MSG_REQUEST_POINT_INFO message to request terrain characteristics for a specified point in the viewing range.

The function call is **upd_req_point()**. The function does the following:

- Makes sure the req_point flag in p_flea_out is TRUE. This flag is set to TRUE if the user selects the R ("request point info") option on the Flea Switches menu.
- Resets the req_point flag to FALSE.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Builds the MSG_REQUEST_POINT_INFO message and puts it in the packet.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: flea_error_print

Parameters: none

Returns: none

2.7.4.17 upd_req_lrange

The upd_req_lrange function generates a MSG_REQUEST_LASER_RANGE message to request laser range depth data for a specified position on the screen.

The function call is **upd_req_lrange()**. The function does the following:

- Makes sure the req_lrange flag in p_flea_out is TRUE. This flag is set to TRUE if the user selects the 1 ("request laser range") option on the Flea Switches menu.
- Resets the req_lrange flag to FALSE.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Builds the MSG_REQUEST_LASER_RANGE message and puts it in the packet.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: flea_error_print

Parameters: none

Returns: none

2.7.4.18 upd_view_mag

The upd_view_mag function generates a MSG_VIEW_MAGNIFICATION message to change the field-of-view parameters for a specified viewport.

The function call is **upd_view_mag()**. The function does the following:

- Makes sure view_mag_flag in p_flea_out is TRUE. This flag is set to TRUE if the user selects the 5 ("magnification") option on the Flea Switches menu, or any of the magnification options on the Flea AGPT Switches menu.
- Resets view_mag_flag to FALSE.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Copies the data to the MSG_VIEW_MAGNIFICATION message and puts it in the packet.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: bcopy
flea_error_print

Parameters: none

Returns: none

2.7.4.19 upd_subsys_mode

The upd_subsys_mode function generates a MSG_SUBSYS_MODE message to change subsystem-specific parameters such as color table and fade value.

The function call is **upd_subsys_mode()**. The function does the following:

- Makes sure subsys_mode_flag in p_flea_out is TRUE. This flag is set to TRUE if the user selects the 6 ("subsys mode") option from the Flea Switches menu. It can also be set by selecting H ("Increase Visibility"), I ("Decrease Visibility"), V ("Single/Dual Vpt Toggle"), or any of the sky color options from the Flea AGPT Switches menu.
- Resets subsys_mode_flag to FALSE.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Copies the data to the MSG_SUBSYS_MODE message and puts it in the packet.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: bcopy
flea_error_print

Parameters: none

Returns: none

2.7.4.20 upd_viewport_up

The upd_viewport_up function generates a MSG_VIEWPORT_UPDATE message to turn a viewport on or off, or to change the viewport's alternate mode (e.g., thermal white hot).

The function call is **upd_viewport_up()**. The function does the following:

- Makes sure `vpt_update_flag` in `p_flea_out` is TRUE. This flag is set to TRUE if the user selects the 7 ("viewport upd") option from the Flea Switches menu. It is also set by selecting F ("Increase Brightness"), G ("Decrease Brightness") or any sky color option from the Flea AGPT Switches menu.
- Resets `vpt_update_flag` to FALSE.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Copies the data to the `MSG_VIEWPORT_UPDATE` message and puts it in the packet.

The function calls `flea_error_print` if the message packet does not contain enough room for the new message.

Called By: `flea_encode_data`

Routines Called: `bcopy`
`flea_error_print`

Parameters: `none`

Returns: `none`

2.7.4.21 **upd_send_stop**

The `upd_send_stop` function generates a `MSG_CIG_CTL` message with the control code set to `C_STOP`. This stops the Flea exercise.

The function call is **upd_send_stop()**. The function does the following:

- Makes sure the stop flag in `p_flea_out` is TRUE. This flag is set to TRUE if the user selects the z ("stop Flea") option on the Flea main menu.
- Resets the stop flag to FALSE.
- Increments the packet count (size).
- Builds the message header and puts it in the packet.
- Builds the `MSG_CIG_CTL` message and puts it in the packet.
- Sets `go_fly_flag` to FALSE.
- Sets the `start_flea` flag in `p_flea_out` to FALSE.
- Resets the `frame_count` to 0.

The function calls `flea_error_print` if the message packet does not contain enough room for the new message.

Called By: `flea_encode_data`

Routines Called: `flea_error_print`

Parameters: none

Returns: none

2.7.4.22 **flea_error_print**

The `flea_error_print` function outputs a "No room in packet" error message to stdout. This function is called whenever one of the `upd_*` functions determines that the message it needs to send cannot fit in the current frame's message packet.

The function call is `flea_error_print(err_str)`, where `err_str` is additional text to be output with the error message. This character string usually identifies the message currently being processed.

Called By:

- `upd_add_static_veh`
- `upd_chord_fired`
- `upd_clouds`
- `upd_dynamic_matrix`
- `upd_lt_state`
- `upd_matrix_values`
- `upd_ppm`
- `upd_rem_static_veh`
- `upd_req_agl`
- `upd_req_lrange`
- `upd_req_point`
- `upd_rotation_values`
- `upd_round_fired`
- `upd_send_dynamic`
- `upd_send_stop`
- `upd_show_eff`
- `upd_subsys_mode`
- `upd_view_flags`
- `upd_view_mag`
- `upd_viewport_up`

Routines Called: `printf`

Parameters: `char` `err_str[]`

Returns: none

2.7.4.23 **put_in_hdr**

The `put_in_hdr` function puts a message's type and size into a data message header, in the format required for the Simulation Host message packet.

The function call is **put_in_hdr(ptr, type, code)**, where:

ptr is a pointer to the message
type is the message type
code is the message code

put_in_hdr does the following:

- Sets the message type in the header to the type specified.
- Sets the message length in the message header to the size (in bytes) of the specified code.
- Increments the pointer to point to the next message.

The function returns the *ptr* specified in the call.

The **put_in_hdr** function is not currently used; each **upd_*** function generates its own data message header.

Called By: none

Routines Called: none

Parameters:	INT_4	*ptr
	INT_2	type
	INT_2	code

Returns: ptr

2.7.4.24 upd_flea_vehicles

The **upd_flea_vehicles** function generates **MSG_TF_VEHICLE_POS** and **MSG_OTHERVEH_STATE** messages for vehicles in the Flea exercise. This function also generates **MSG_TF_INIT_HDR** and **MSG_TF_INIT_PT** messages for vehicles if their terrain feedback processing has not been enabled.

The function call is **upd_flea_vehicles()**. The function does the following for each Flea vehicle:

- If the vehicle's **in_use_flag** and **tf_init_ready_flag** are TRUE (set using the **i** ("init veh") option on the Vehicle Control menu), but its **tf_init_sent_flag** is FALSE:
 - Builds a message header and puts it in the outgoing message packet.
 - Builds a **MSG_TF_INIT_HDR** message and puts it in the message packet.
 - For each terrain feedback point on the vehicle, builds a **MSG_TF_INIT_PT** message and puts it in the message packet.
 - Sets the vehicle's **tf_init_sent_flag** to TRUE.
- If the vehicle's **in_use_flag** is TRUE and the vehicle index is 1:
 - Builds a **MSG_TF_VEHICLE_POS** message and header and puts them in the message packet.

- Builds a MSG_OTHERVEH_STATE message and header and puts them in the message packet.

The function always returns 0.

Called By: flea_encode_data

Routines Called: bcopy
 printf (in debug mode only)

Parameters: none

Returns: 0

2.7.4.25 fire_round

The fire_round function generates a MSG_ROUND_FIRED message to tell Ballistics that a round has been fired in the Flea exercise.

The function call is **fire_round(p_msg)**, where *p_msg* is a pointer to the MSG_ROUND_FIRED message.

fire_round does the following:

- Builds the message header.
- Puts the round data (primarily taken from p_flea_out) into the MSG_ROUND_FIRED message. The calling function puts the message into the message packet.

The function returns the message pointer as *p_msg*.

Called By: upd_auto_fire
 upd_round_fired

Routines Called: none

Parameters: MSG_ROUND_FIRED *p_msg

Returns: p_msg

2.7.4.26 process_chord

The process_chord function generates a MSG_PROCESS_CHORD message to tell Ballistics to process a chord's intersection with the database.

The function call is **process_chord(p_msg)**, where *p_msg* is a pointer to the MSG_PROCESS_CHORD message.

process_chord does the following:

- Builds the message header.
- Puts the chord data (primarily taken from *p_flea_out*) into the MSG_PROCESS_CHORD message. The calling function puts the message into the message packet.

The function returns the message pointer as *p_msg*.

Called By:	upd_chord_fired	
Routines Called:	none	
Parameters:	MSG_PROCESS_CHORD	*p_msg
Returns:	p_msg	

2.7.4.27 **process_round**

The **process_round** function generates a MSG_PROCESS_ROUND message to tell Ballistics that a round has been fired in the Flea exercise.

The function call is **process_round(p_msg)**, where *p_msg* is a pointer to the MSG_PROCESS_ROUND message.

process_round does the following:

- Builds the message header.
- Puts the round data into the MSG_PROCESS_ROUND message. The calling function puts the message in the message packet.

The function returns the message pointer as *p_msg*.

Called By:	upd_auto_fire upd_round_fired	
Routines Called:	none	
Parameters:	MSG_PROCESS_ROUND	*p_msg
Returns:	p_msg	

2.7.4.28 **cancel_round**

The **cancel_round** function generates a MSG_CANCEL_ROUND message to tell Ballistics to delete a fired round.

The function call is **cancel_round(p_msg)**, where *p_msg* is a pointer to the MSG_CANCEL_ROUND message.

cancel_round does the following:

- Builds the message header.
- Puts the round data into the MSG_CANCEL_ROUND message. The calling function puts the message into the message packet.

The function returns the message pointer as *p_msg*.

Called By: upd_round_fired

Routines Called: none

Parameters: MSG_CANCEL_ROUND *p_msg

Returns: p_msg

2.7.4.29 **upd_sio_write**

The **upd_sio_write** function generates a MSG_SIO_WRITE message to write to a serial input/output device.

The function call is **upd_sio_write()**. The function does the following:

- Makes sure **sio_flag** is TRUE. This flag is set to TRUE by **get_sio_data** if the user selects the Z ("Write to Video Mux") option from the Flea AGPT Switches menu.
- Sets **sio_flag** to FALSE.
- Puts the message header in the packet.
- Copies the data to the MSG_SIO_WRITE message and puts it in the packet.

Called By: flea_encode_data

Routines Called: bcopy

Parameters: none

Returns: none

2.7.4.30 upd_lt_state

The upd_lt_state function generates a MSG_LT_STATE message to change the parameters used to generate local terrain messages.

The function call is **upd_lt_state()**. The function does the following:

- Makes sure lt_state_flag is TRUE. This flag is set to TRUE if the user selects option 4 ("change lt state") on the Flea Switches menu.
- Sets lt_state_flag to FALSE.
- Increments the packet count (size).
- Puts the message header in the packet.
- Builds the MSG_LT_STATE message and puts it in the packet.

The function calls flea_error_print if the message packet does not contain enough room for the new message.

Called By: flea_encode_data

Routines Called: flea_error_print

Parameters: none

Returns: none

2.7.4.31 send_gun_overlay

The send_gun_overlay function generates a MSG_GUN_OVERLAY message to change the components of the M1 or M2 gunner's overlay.

The function call is **send_gun_overlay()**. The function does the following:

- Makes sure send_gun_stat is TRUE. This flag is set to TRUE if the user selects the g option ("gun barrel update") on the Flea Switches menu.
- Puts the message header in the packet.
- Builds the MSG_GUN_OVERLAY message and puts it in the packet.

Called By: flea_encode_data

Routines Called: cos
sin
TORAD

Parameters: none

Returns: none

2.7.4.32 send_ammo_define

The send_ammo_define function generates a MSG_AMMO_DEFINE message to define ammunition maps.

The function call is **send_ammo_define()**. The function does the following:

- Makes sure flea_ammo_define_flag is TRUE. This flag is set to TRUE if the user selects the 9 option ("ammo define msg") on the Flea Switches menu.
- Sets flea_ammo_define_flag to FALSE.
- Increments the packet count (size).
- Puts the message header in the packet.
- Copies the data to the MSG_AMMO_DEFINE message and puts it in the packet.

Called By: flea_encode_data

Routines Called: none

Parameters: none

Returns: none

2.7.5 flea.c

The functions in the flea.c CSU form the main driver for the Flea exercise. These functions are:

- flea
- flea_io_task
- flea_getchar
- flea_dummy_getchar
- flea_IO_mode
- flea_IO_on
- flea_IO_off
- flea_initialized
- scratch_flea
- flea_printf
- flea_cleanup
- flea_io_task_cleanup

2.7.5.1 flea

The flea function is the main driver for the Flea emulation. The flea task is created and started by Task Initialization CSC. flea then suspends itself until a message is posted to the FLEA_MB mailbox. This message is posted by scratch_flea, which is called by gossip_tick if the system user selects the f ("enter Flea menu") option from the Gossip main menu.

The function call is `flea()`. flea does the following:

- Calls `poll_shutdown` to see if a system shutdown has been initiated.
- Waits for a message to be posted to the FLEA_MB mailbox.
- Allocates memory for the `p_flea_in` and `p_flea_out` structures.
- Opens the Flea console port. (This is the station through which the user interacts with the system. It can be specified through an option on the Gossip main menu.)
- Prompts the user for the initial viewpoint position; uses the entered coordinates to set `flea_vppos.x`, `y`, and `z`.
- Prompts the user for the initial viewpoint orientation; uses the entered values to set heading, pitch, and roll.
- If the frame count is 0, initializes various variables and structures.
- If the `start_flea` flag in `p_flea_out` is TRUE (indicating that the user has *not* requested to stop Flea):
 - Calls `poll_shutdown` to see if a system shutdown has been initiated.
 - Initializes the `p_flea_out` flags used to indicate what changes have been requested by the user.
 - Initializes other variables in `p_flea_out`, including transformation, rotation, branch, database traversal, and process chord values.
 - Allocates memory for other vehicle state messages.
 - Allocates memory for the Flea simulated vehicles data.
 - If flea has just started, calls `OPEN_FLEA_DATA` to establish the CIG-Flea communications path.
 - If the flea mode is set to `FLEA_ENET` (use Ethernet):
 - * Opens the `fleanet.cfg` file.
 - * Gets the configuration file name.
 - * Gets the host's Ethernet address.
 - * Sets `flea_init_slave_cig_G` to TRUE.
 - * Calls `EXCHANGE_FLEA_DATA` to synchronize with a Slave CIG.
 - If the flea mode is anything other than `FLEA_ENET`, calls `flea_init_cig_sw` to configure Flea.
 - Initializes `flea_esifa_load_wanted_G` and `flea_color_cfg_wanted_G` to FALSE.
 - Generates a `MSG_CIG_CTL` message to put the CIG into the simulation state.
 - Calls `EXCHANGE_FLEA_DATA` to exchange packets with the CIG.
 - Sets the running flag to TRUE.
 - Calls `tick_init` to get the user's command.
 - Sets the `*flea_menu` function pointer to `tick`.
 - Calls `sc_tresume` to wake up `flea_io_task`. (`flea_io_task` uses the `*flea_menu` function pointer to call the appropriate menu function during the Flea exercise.)

Once the Flea simulation is running, flea performs the following loop:

- Calls `poll_shutdown` to see if a system shutdown has been initiated.
- If running in absolute playback mode, calls `flea_abs_playback`. This mode is set if the user selects the `b` ("begin absolute playback") option from the Flea Script menu.
- If the user is driving the exercise:
 - Calls `flea_update_pos` to update the position of the simulated vehicle.
 - Calls `flea_decode_data` to process all CIG-to-Flea messages.
 - Calls `flea_encode_data` to process all Flea-to-CIG messages.
- Uses the `EXCHANGE_FLEA_DATA` macro to exchange message packets with the CIG each frame.

When the user requests that the exercise be stopped, flea does the following:

- Frees all allocated memory.
- Closes the console port.
- Sets the running flag to `FALSE`.

flea returns -1 if it cannot allocate memory for `p_flea_in` or `p_flea_out`. It exits with a 1 if it cannot open the Flea console port or the Ethernet configuration file.

Called By: none (task is created and started at initialization time)

Routines Called:

- `calloc`
- `close`
- `EXCHANGE_FLEA_DATA`
- `exit`
- `fgets`
- `flea_abs_playback`
- `flea_decode_data`
- `flea_encode_data`
- `flea_init_cig_sw`
- `flea_update_pos`
- `fopen`
- `fprintf`
- `free`
- `malloc`
- `open`
- `OPEN_FLEA_DATA`
- `poll_shutdown`
- `printf`
- `read_tty`
- `rewind`
- `rt_pend`
- `sc_tresume`
- `scanf`
- `sscanf`
- `strcmp`
- `strcpy`
- `tick_init`

Parameters: none

Returns: -1

2.7.5.2 **flea_io_task**

The **flea_io_task** function drives Flea input/output during a Flea exercise, by calling the function responsible for the menu currently displayed. The menu function then gets and processes the keystroke entered by the user. **flea_io_task** is created and started by the Task Initialization CSC, then suspends itself until woken up by **flea**.

The function call is **flea_io_task()**. **flea_io_task** does the following:

- Suspends itself until woken up by **flea** after Flea mode has been selected and configuration is complete.
- Calls **poll_shutdown** to see if a system shutdown has been initiated.
- Uses the ***flea_menu** function pointer to call the active menu function to get the user's entry. The function called by ***flea_menu** is the function that handles the menu currently displayed. When Flea first starts, this is the tick function, which processes the Flea main menu. The ***flea_menu** pointer is reset as the user traverses the Flea menus.
- Pauses for ten milliseconds, then calls ***flea_menu** again. Repeats this loop until a shutdown is detected.

Called By: none (task is created and started at initialization time)

Routines Called: ***flea_menu**
poll_shutdown
sc_delay
sc_tinquiry
sc_tsuspend

Parameters: none

Returns: none

2.7.5.3 **flea_getchar**

The **flea_getchar** function returns the key pressed by the Flea operator. It returns 0 if no key was pressed. All Flea functions that process user input use **flea_getchar** to determine the key pressed by the user.

The function call is **flea_getchar()**.

This function is called indirectly through the ***flea_getc** function pointer. The **flea_IO_on** function sets ***flea_getc** to **flea_getchar** when Flea is invoked. When Flea mode is stopped,

`flea_IO_off` sets `*flea_getc` to `flea_dummy_getchar`; this effectively disables the Flea user interface and returns keyboard control to Gossip.

Called By:	<code>flea_agpt_locations</code>	(through <code>*flea_getc</code>)
	<code>flea_agpt_switches</code>	(through <code>*flea_getc</code>)
	<code>flea_atp</code>	(through <code>*flea_getc</code>)
	<code>flea_bal_opts</code>	(through <code>*flea_getc</code>)
	<code>flea_graphics_test</code>	(through <code>*flea_getc</code>)
	<code>flea_IO_on</code>	(through <code>*flea_getc</code>)
	<code>flea_switches</code>	(through <code>*flea_getc</code>)
	<code>flea_veh_control</code>	(through <code>*flea_getc</code>)
	<code>tick</code>	(through <code>*flea_getc</code>)
	<code>tick_ppm</code>	(through <code>*flea_getc</code>)
	<code>tick_script</code>	(through <code>*flea_getc</code>)

Routines Called: `read_tty`

Parameters: none

Returns: `key`
0

2.7.5.4 `flea_dummy_getchar`

The `flea_dummy_getchar` function always returns 0. This function is used in place of `flea_getchar` when the user exits Flea mode; this effectively disables the Flea user interface and returns keyboard control to Gossip. (When a Flea user interface function receives a 0 return value, it assumes no key was pressed and therefore does not process it.)

The function call is `flea_dummy_getchar()`.

This function is called indirectly through the `*flea_getc` function pointer. The `flea_IO_on` function sets `*flea_getc` to `flea_getchar` when Flea is invoked. When Flea mode is stopped, `flea_IO_off` sets `*flea_getc` to `flea_dummy_getchar`.

Called By:	<code>flea_agpt_locations</code>	(through <code>*flea_getc</code>)
	<code>flea_agpt_switches</code>	(through <code>*flea_getc</code>)
	<code>flea_atp</code>	(through <code>*flea_getc</code>)
	<code>flea_bal_opts</code>	(through <code>*flea_getc</code>)
	<code>flea_graphics_test</code>	(through <code>*flea_getc</code>)
	<code>flea_IO_off</code>	(through <code>*flea_getc</code>)
	<code>flea_switches</code>	(through <code>*flea_getc</code>)
	<code>flea_veh_control</code>	(through <code>*flea_getc</code>)
	<code>tick</code>	(through <code>*flea_getc</code>)
	<code>tick_ppm</code>	(through <code>*flea_getc</code>)
	<code>tick_script</code>	(through <code>*flea_getc</code>)

Routines Called: none

Parameters: none

Returns: 0

2.7.5.5 **flea_IO_mode**

The **flea_IO_mode** function returns TRUE if Flea input/output mode is enabled, and FALSE if it is not. This response is used by tick to determine whether it should call **update_menu_header** to update the simulated vehicle statistics at the top of the Flea console screen.

Flea input/output mode is enabled (by **flea_IO_on**) when the user selects the **f** ("enter Flea menu") option from the Gossip main menu. It is disabled (by **flea_IO_off**) when the user exits from Flea. The boolean variable used to maintain the status is **io_enabled**.

The function call is **flea_IO_mode()**.

Called By: tick

Routines Called: none

Parameters: none

Returns: 1 (TRUE)
0 (FALSE)

2.7.5.6 **flea_IO_on**

The **flea_IO_on** function enables Flea input/output mode. This function is called when the user selects the **f** ("enter Flea menu") option from the Gossip main menu.

The function call is **flea_IO_on()**. The function does the following:

- Sets the ***flea_getc** function pointer to **flea_getchar**. This allows the Flea user interface functions to get the keystroke entered by the user.
- Sets the **io_enabled** variable to TRUE. This variable is used by **flea_IO_mode**.

Called By: gossip_tick

Routines Called: printf (in debug mode only)

Parameters: none

Returns: none

2.7.5.7 flea_IO_off

The flea_IO_off function disables Flea input/output mode. This function is called when the user stops the Flea exercise (or returns to Gossip but leaves the Flea simulation running).

The function call is **flea_IO_off()**. The function does the following:

- Sets the *flea_getc function pointer to flea_dummy_getchar. This effectively disables the Flea user interface and returns keyboard control to Gossip.
- Sets the io_enabled variable to FALSE. This variable is used by flea_IO_mode.

Called By: gossip
tick

Routines Called: printf (in debug mode only)

Parameters: none

Returns: none

2.7.5.8 flea_initialized

The flea_initialized function returns TRUE if a Flea simulation is active, and FALSE if it is not. It examines the running flag, which is set to TRUE by flea after the Flea configuration file has been processed and the simulation state has been entered. gossip_tick calls the flea_initialized function to verify that the Flea simulation started successfully.

The function call is **flea_initialized()**.

Called By: gossip_tick

Routines Called: none

Parameters: none

Returns: 1 (TRUE)
0 (FALSE)

2.7.5.9 scratch_flea

The `scratch_flea` function posts a message to the `FLEA_MB` mailbox to wake up the flea function. `scratch_flea` is called when the user selects the `f` ("enter Flea menu") option from the Gossip main menu. If a Flea simulation is already running (because the user exited to Gossip during the exercise and is now returning), `scratch_flea` does nothing.

The function call is `scratch_flea()`.

Called By: `gossip_tick`

Routines Called: `rt_post`

Parameters: `none`

Returns: `none`

2.7.5.10 flea_printf

The `flea_printf` function is not currently used.

2.7.5.11 flea_cleanup

The `flea_cleanup` function deallocates the resources owned by the flea task. This function is called if a system shutdown is requested by the Gossip user. The function is called via the `*task_cleanup` function pointer, which points to the cleanup routine's name in the task table.

The function call is `flea_cleanup()`.

The function returns 1 if successful, or 0 if an error occurred.

Note: This function is not yet implemented. At the current time, it simply returns a 1 if called.

Called By: `poll_shutdown` (through `*task_cleanup`)

Routines Called: `none`

Parameters: `none`

Returns: `0`

2.7.5.12 flea_io_task_cleanup

The `flea_io_task_cleanup` function deallocates the resources owned by `flea_io_task`. This function is called if a system shutdown is requested by the Gossip user. The function is called via the `*task_cleanup` function pointer, which points to the cleanup routine's name in the task table.

The function call is `flea_io_task_cleanup()`.

The function returns 1 if successful, or 0 if an error occurred.

***Note:** This function is not yet implemented. At the current time, it simply returns a 1 if called.*

Called By: poll_shutdown (through `*task_cleanup`)

Routines Called: none

Parameters: none

Returns: 0
1

2.7.6 flea_agl_terrain_follow.c

The `flea_agl_terrain_follow` function adjusts the Flea vehicle's altitude (`flea_vppos.z`) each frame using a desired altitude entered by the user. The AGL terrain follow option is set by selecting the `g` ("toggle agl ground follow") option from the Flea Ballistics menu. The user is prompted for the desired height above ground level.

The function call is `flea_agl_terrain_follow()`. The function does the following:

- Makes sure the `agl_terrain_follow` flag is set.
- If the AGL value returned by the real-time software (via the `MSG_AGL` message) is less than 0, sets `flea_vppos.z` to its old value plus the desired AGL entered by the user on the Flea Ballistics menu.
- If the returned AGL value is 0 or greater, sets `flea_vppos.z` to its old value plus the average of the desired and returned values (desired AGL minus returned AGL, divided by 2).

The function always returns 0.

Called By: flea_update_pos

Routines Called: none

Parameters: none

Returns: 0

2.7.7 flea_agpt_locations.c

The functions in the flea_agpt_locations.c CSU let the user run special acceptance tests and invoke moving model demonstrations. These functions are:

- flea_agpt_locations
- flea_agpt_locations_main_menu
- new_pos_orient
- update_dyn_demo

2.7.7.1 flea_agpt_locations

The flea_agpt_locations function processes the user's sections from the Flea AGPT Locations menu. This function is called (via the *flea_menu function pointer) when the user selects the ("AGPT locations") option from the Flea main menu.

The function call is **flea_agpt_locations()**. The function uses the *flea_getc function pointer to get the keystroke input by the user, then processes it.

The following table identifies the options supported by flea_agpt_locations, and shows the steps it performs to process each one. For each acceptance test selected by the user, flea_agpt_locations calls new_pos_orient with the vehicle coordinates used to access the portion of the database that contains the selected test.

The Acceptance Test Scenes menu is displayed by flea_agpt_locations_main_menu. Options flagged with an asterisk are supported by flea_agpt_locations but do not appear on the menu.

Acceptance Test Scenes Menu Option	Processing by flea_agpt_locations
?* Display this menu	Calls flea_agpt_locations_main_menu.
A Triangle	Calls new_pos_orient.
B Chessboards	Calls new_pos_orient.
C Colors	Calls new_pos_orient.
D House	Calls new_pos_orient.
E Textures	Calls new_pos_orient.
F Texture color	Calls new_pos_orient.
G Pyramids	Calls new_pos_orient.
H Impact/Tank	Calls new_pos_orient.
I Forest/Tank	Calls new_pos_orient.
J Tree/Tank	Calls new_pos_orient.
L 50 Cubes	Calls new_pos_orient.
M 1 Polyhedron	If demonstration = 0: calls new_pos_orient; sets skidx, skidy, skidz, htrate, and speed to 0; calls dynamic_demo with frame number set to 0 (to start demo); sets dyn_models to TRUE. Calls update_dyn_demo; calls new_pos_orient.
N 10 Polyhedrons	If demonstration = 0: calls new_pos_orient; sets skidx, skidy, skidz, htrate, and speed to 0; calls dynamic_demo with frame number set to 0 (to start demo); sets dyn_models to TRUE. Calls update_dyn_demo; calls new_pos_orient.
O 20 Polyhedrons	If demonstration = 0: calls new_pos_orient; sets skidx, skidy, skidz, htrate, and speed to 0; calls dynamic_demo with frame number set to 0 (to start demo); sets dyn_models to TRUE. Calls update_dyn_demo; calls new_pos_orient.
P 30 Polyhedrons	If demonstration = 0: calls new_pos_orient; sets skidx, skidy, skidz, htrate, and speed to 0; calls dynamic_demo with frame number set to 0 (to start demo); sets dyn_models to TRUE Calls update_dyn_demo; calls new_pos_orient.
Q 1 pixel line	Calls new_pos_orient.
R 4 Tanks-Text. Area	Calls new_pos_orient.
S 4 Tanks-Untext. Area	Calls new_pos_orient.
X Quit Polyheds	If dyn_models is TRUE: sets demonstration to 0; calls dynamic_demo with frame set to 9999 (to end demo); sets dyn_models to FALSE.
x* Exit	Sets *flea_menu function pointer to tick; sets prompt to "Flea>"; calls tick_main_menu.

Called By: flea_io_task (through *flea_menu)

Routines Called: *flea_getc
 cup
 dynamic_demo
 flea_agpt_locations_main_menu
 isprint
 isspace
 new_pos_orient
 printf
 strcpy
 tick_main_menu
 update_dyn_demo

Parameters: none

Returns: none

2.7.7.2 **flea_agpt_locations_main_menu**

The `flea_agpt_locations_main_menu` function displays the Acceptance Test Scenes menu, which is processed by the `flea_agpt_locations` function. This function is called if the user selects the ("AGPT locations") option from the Flea main menu. It is also called if the user enters ? at the "Flea AGPT Locations>" prompt.

The function call is `flea_agpt_locations_main_menu()`. The function does the following:

- Clears the screen.
- Calls `menu_header` to display the labels for the Flea simulation statistics.
- Displays the Acceptance Test Scenes menu.
- Calls `update_menu_header` to display the current Flea simulation statistics.
- Displays the "Flea AGPT Locations>" prompt.

For a list of the options displayed on the Acceptance Test Scenes menu, see `flea_agpt_locations`.

Called By: flea_agpt_locations
 tick

Routines Called: blank
 cup
 menu_header
 printf
 strcpy
 update_menu_header

Parameters: none

Returns: none

2.7.7.3 new_pos_orient

The `new_pos_orient` function changes the simulated vehicle's coordinates (`flea_vppos.x`, `y`, and `z`) and rotation values (`flea_vprot.x`, `y`, and `z`). This function is called whenever the user requests a test via the Acceptance Tests Scenes menu. It positions the vehicle at a special database location that contains the acceptance test scene.

The function call is `new_pos_orient(x, y, z, rz, rx, ry)`, where:

`x` is the vehicle's x coordinate
`y` is vehicle's y coordinate
`z` is vehicle's z coordinate
`rz` is the vehicle's x rotation value
`rx` is the vehicle's y rotation value
`ry` is vehicle's z rotation value

Called By: `flea_agpt_locations`

Routines Called: none

Parameters:	REAL_4	x
	REAL_4	y
	REAL_4	z
	REAL_4	rz
	REAL_4	rx
	REAL_4	ry

Returns: none

2.7.7.4 update_dyn_demo

The `update_dyn_demo` function sets up the system to run any of the moving model tests listed on the Acceptance Test Scenes menu. This function is called if the user selects any of the "Polyhedron" options.

The function call is `update_dyn_demo(num_models, model_id, numx, numy, numz, spx, spy, spz)`, where:

`num_models` is the number of models in the demo
`model_id` is the model id to be used
`numx` is the number of models in the x direction
`numy` is the number of models in the y direction
`numz` is the number of models in the z direction
`spx` is the spacing between models in the x direction
`spy` is the spacing between models in the y direction

spz is the spacing between models in the z direction

update_dyn_demo sets the given parameters in the demo_ptr structure in p_flea_out.
demo_ptr is used by dynamic_demo, which actually runs the demonstration.

Called By: flea_agpt_locations

Routines Called: none

Parameters:	INT_4	num_models
	INT_4	model_id
	INT_4	numx
	INT_4	numy
	INT_4	numz
	REAL_4	spx
	REAL_4	spy
	REAL_4	spz

Returns: none

2.7.8 flea_agpt_switches.c

The functions in the flea_agpt_switches.c CSU adjust sky color, light intensity, and magnification. These functions are:

- flea_agpt_switches
- set_command_2d
- update_mag
- update_subsys_mode
- update_vpt
- flea_agpt_switches_main_menu
- derror

2.7.8.1 flea_agpt_switches

The flea_agpt_switches function processes the user's selections on the Flea AGPT Switches menu. This menu is used to adjust sky color, light intensity, and magnification. This function is called (via the *flea_menu function pointer) when the user selects the) ("AGPT switches") option from the Flea main menu.

The function call is flea_agpt_switches(). The function uses the *flea_getc function pointer to get the keystroke entered by the user, then processes it.

The following table lists the options supported by flea_agpt_switches, and shows the steps it uses to process each one. This menu is displayed by flea_agpt_switches_main_menu. Options flagged with an asterisk are supported by flea_agpt_switches but do not appear on the menu.

AGPT Switches Menu Option	Processing by flea_agpt_switches
?* Display this menu	Calls flea_agpt_switches_main_menu.
(Thermal Toggle	Changes sky_color_value from 3 (white hot) to 4 (black hot) or vice versa; calls update_subsys_mode.
A Blue Sky	Sets sky_color_value (0), light_intensity, p_flea_out->branch_value[0], and vis_range; calls update_subsys_mode; calls update_vpt.
B Light Grey Sky	Sets sky_color_value (1), light_intensity, p_flea_out->branch_value[0], and vis_range; calls update_subsys_mode; calls update_vpt.
C Dark Grey Sky	Sets sky_color_value (2), light_intensity, p_flea_out->branch_value[0], and vis_range; calls update_subsys_mode; calls update_vpt.
D White Hot Sky	Sets sky_color_value (3), light_intensity, and vis_range; calls update_subsys_mode; sets p_flea_out->branch_value[0]; calls update_vpt.
E Black Hot Sky	Sets sky_color_value (4), light_intensity, and vis_range; calls update_subsys_mode; sets p_flea_out->branch_value[0]; calls update_vpt.
F Increase Brightness	If maximum light_intensity already reached: calls derror. Else: changes light intensity; sets viewport on; sets alternate mode on if sky color is thermal; sets vpt_update_flag to TRUE; changes intensity in color table; sets subsys_mode_flag to TRUE.
G Decrease Brightness	If minimum light_intensity already reached: calls derror. Changes light intensity. If minimum light intensity now reached and sky color is blue or grey: sets viewport to off; sets vpt_update_flag to TRUE. Else: sets viewport on; sets alternate mode on if sky color is thermal; sets vpt_update_flag to TRUE; changes intensity in color table; sets subsys_mode_flag to TRUE.
H Increase Visibility	If maximum visibility already reached, calls derror. Else: changes visibility range; puts new fade value in p_flea_out; sets subsys_mode_flag to TRUE.
I Decrease Visibility	If minimum visibility already reached, calls derror. Else: changes visibility range; puts new fade value in p_flea_out; sets subsys_mode_flag to TRUE.
J WBG 4x Reticle	Calls set_command_2d (command=1); sets reticle_on to TRUE.
K WBG 12x Reticle	Calls set_command_2d (command=2); sets reticle_on to TRUE.
L EMES 12x Reticle	Calls set_command_2d (command=3); sets reticle_on to TRUE.
M PERI 2x Reticle	Calls set_command_2d (command=4); sets reticle_on to TRUE.

N	PERI 8x Reticle	Calls set_command_2d (command=5); sets reticle_on to TRUE.
O	2D Brighter	If maximum 2-D intensity already reached: calls derror. Else: increments bright_2d; sets new_2d_color; calls set_command_2d (command=12).
P	2D Dimmer	If minimum 2-D intensity already reached: calls derror. Else: decrements bright_2d; sets new_2d_color; calls set_command_2d (command=12).
Q	2x Magnification	Calls update_mag; sets magnification_on to TRUE.
R	4x Magnification	Calls update_mag; sets magnification_on to TRUE.
S	8x Magnification	Calls update_mag; sets magnification_on to TRUE.
T	12x Magnification	Calls update_mag; sets magnification_on to TRUE.
U	Moving Model Menu	If demonstration = 0 (new demo): initializes flea_vprot.x, flea_vprot.y, flea_vprot.z, skidx, skidy, skidz, htrate, and speed to 0; calls dynamic_demo with frame number set to 0; calls derror to tell user to select U again to change options (via model_demo). If demonstration = 2 (demo in progress): calls model_demo.
V	Single/Dual Vpt Toggle	If dual_mode_on is TRUE: sets dual_mode_on to FALSE; sets p_flea_out->subsys_mode.special_use to 0; sets p_flea_out->subsys_mode_flag to TRUE; sets p_flea_out->branch_value[0] to 0. If dual_mode_on is FALSE: sets dual_mode_on to TRUE; sets p_flea_out->subsys_mode.special_use to 0x200; sets p_flea_out->subsys_mode_flag to TRUE; sets p_flea_out->branch_value[0] to 2.
W	Remove 2d Reticle	If reticle_on is TRUE, sets reticle_on to FALSE; calls set_command_2d (command=15).
X	No Magnification	If magnification_on is TRUE, sets magnification_on to FALSE; calls update_mag to reset lod and fov.
x*	Exit	Sets *flea_menu function pointer to tick; sets prompt to "Flea>"; calls tick_main_menu.
Y	Aliasing Model (2D)	Calls set_command_2d (command=6); sets reticle_on to TRUE.
Z	Write to Video Mux	Calls get_sio_write_data.

Called By: flea_io_task (through *flea_menu)

Routines Called: *flea_getc
cup
derror
dynamic_demo
flea_agpt_switches_main_menu
get_sio_write_data
isprint
isspace

```

model_demo
printf
set_command_2d
strcpy
tick_main_menu
update_mag
update_subsys_mode
update_vpt

```

Parameters: none

Returns: none

2.7.8.2 set_command_2d

The `set_command_2d` function sets up data to be sent to the 2-D overlay task via a `MSG_PASS_ON` message. This function is called if the user selects "Aliasing Model (2D)," "2D Dimmer," "2D Brighter," or any of the "Reticle" options from the Flea AGPT Switches menu.

The function call is `set_command_2d(subsys, cmd)`, where:

subsys is the subsystem to which the message is to be sent (currently always 0 for the 2-D overlay processor)
cmd is the command to be sent

`set_command_2d` does the following:

- Checks to make sure the subsystem has a Force board.
- Puts the command into the subsystem's element in the `command_2d[]` array, for processing by `update_agpt_2d`.
- Sets `p_flea_out->sub_2d_io[subsystem]` to TRUE. When this flag is set, `flea_encode_data` calls `update_agpt_2d` to generate the `MSG_PASS_ON` message. (`flea_encode_data` uses the `type_update_2d` variable to determine whether to call `update_agpt_2d` or `update_2d`. `flea_agpt_switches` sets `type_update_2d` to 1, which causes `flea_encode_data` to select the AGPT version.)
- Sets the subsystem's flag in the `gsp_io_flag[]` array to TRUE. This tells the `mpvideo_send_req` function (in the MPV Interface CSC) that there is a command to be sent to the GSP in this subsystem.

Called By: `flea_agpt_switches`

Routines Called: none

Parameters:	INT_4	subsys
	INT_4	cmd

Returns: none

2.7.8.3 update_mag

The `update_mag` function puts new viewport magnification values into the `p_flea_out` buffer for processing by the encode routines. This function is called if the user selects any of the magnification options on the Flea AGPT Switches menu.

The function call is `update_mag(node, lod, fovi, fovj)`, where:

node is the configuration node with the viewport parameters to be modified

lod is the new level of detail

fovi is the new horizontal field-of-view angle

fovj is the new vertical field-of-view angle

`update_mag` puts the new parameters into `p_flea_out` and sets `view_mag_flag` to TRUE. The message to make the change is generated by `upd_view_mag`.

Called By: fleas_agpt_switches

Routines Called: none

Parameters:	INT_4	node
	REAL_4	lod
	REAL_4	fovi
	REAL_4	fovj

Returns: none

2.7.8.4 update_subsys_mode

The `update_subsys_mode` function puts new color table and fade value data into the `p_flea_out` buffer for processing by the encode routines. This function is called when the user selects a new sky color via the Flea AGPT Switches menu.

The function call is `update_subsys_mode(subsys, color, fade, special)`, where:

subsys is the subsystem (backend) id

color is the color table number

fade is the fade value

special is the new TX mode

`update_subsys_mode` puts the new parameters into `p_flea_out` and sets `subsys_mode_flag` to TRUE. The message to make the change is generated by `upd_subsys_mode`.

Called By: fleas_agpt_switches

Routines Called: none

Parameters:	HWORD	subsys
	BYTE	color
	BYTE	fade
	WORD	special

Returns: none

2.7.8.5 update_vpt

The `update_vpt` function puts new viewport parameters into the `p_flea_out` buffer for processing by the encode routines. This function is called if the user selects a new sky color from the Flea AGPT Switches menu.

The function call is `update_vpt(id, on, alt, mod)`, where:

id is the viewport id

on is 0 to turn the viewport on, or 1 to turn it off

alt is 0 to disable alternate mode, or 1 to enable it

mod is 0 to disable the alternate mode modifier, or 1 to enable it

`update_vpt` puts the new parameters into `p_flea_out` and sets `vpt_update_flag` to TRUE. The message to change the viewport is generated by `upd_viewport_up`.

Called By: fleas_agpt_switches

Routines Called: none

Parameters:	INT_2	id
	BYTE	on
	BYTE	alt
	BYTE	mod

Returns: none

2.7.8.6 fleas_agpt_switches_main_menu

The `fleas_agpt_switches_main_menu` function displays the Flea AGPT Switches menu. This function is called if the user selects the) ("AGPT switches") option from the Flea main menu. It is also called if the user enters ? at the "Flea AGPT Switches>" prompt.

The function call is `fleas_agpt_switches_main_menu()`. The function does the following:

- Clears the screen.
- Calls menu_header to display the labels for the Flea simulation statistics.
- Displays the current sky color and fade values.
- Displays the menu.
- Calls update_menu_header to display the current Flea simulation statistics.
- Displays the "Flea AGPT Switches>" prompt.

For a list of the options on the Flea AGPT Switches menu, see `flea_agpt_switches`.

Called By: flea_agpt_switches
tick

```
Routines Called:    blank  
                   cup  
                   menu_header  
                   printf  
                   strcpy  
                   update_menu_header
```

Parameters: none

Returns: `none`

2.7.8.7 derror

The `derror` function displays error messages and prompts for `flea_agpt_switches`.

The function call is **derror(s)**, where *s* is the text to be displayed. **derror** also sets **user_error** to **TRUE**; this variable is not used elsewhere.

Called By: flea_agpt_switches

Routines Called: cup
printf

Parameters: **char** **s[]**

Returns: none

2.7.9 flea_atp.c

The functions in the `flea_atp.c` CSU are used to run calibration acceptance tests under Flea. These functions are:

- flea_atp
- flea_atp_main_menu

2.7.9.1 flea_atp

The `flea_atp` function lets the Flea user run the acceptance tests that use the calibration database. This function is called via the `*flea_menu` function pointer when the user selects the \$ ("DBCALIB locations" - database calibration locations) option from the Flea main menu.

The function call is `flea_atp()`. `flea_atp` prompts the user for a selection and uses the `*flea_getc` function pointer to get the entered keystroke.

For each test selected by the user, `flea_atp` sets the simulated vehicle's coordinates (`flea_vppos.x`, `y`, and `z`) to the database location used for that test. The vehicle's rotation values (`flea_vprot.x`, `y`, and `z`), `skidx`, `skidy`, `skidz`, `htrate`, and `speed` are all set to 0.

The following table lists the options supported by `flea_atp`, and shows the steps it performs to process each one. This menu is displayed by `flea_atp_main_menu`.

Acceptance Test Menu Option	Processing by flea_atp
? help	Calls <code>flea_atp_main_menu</code> .
0 Populated Area	Sets the simulated vehicle's current position.
1 Depth Complexity	Sets the simulated vehicle's current position.
2 Color Resolution	Sets the simulated vehicle's current position.
3 Full Perspective Texture	Sets the simulated vehicle's current position.
4 Level of Detail	Sets the simulated vehicle's current position.
6 Occulting Levels	Sets the simulated vehicle's current position.
7 Polygon Throughput	Sets the simulated vehicle's current position.
8 Texture with Transparency	Sets the simulated vehicle's current position.
9 Polygon Test Pattern	Sets the simulated vehicle's current position.
x Exit	Sets <code>*flea_menu</code> function pointer to tick; calls <code>strcpy</code> to display "Flea>" prompt; calls <code>tick_main_menu</code> .

Called By: `flea_io_task` (through `*flea_menu`)

Routines Called: `*flea_getc`
 `cup`
 `flea_atp_main_menu`
 `isprint`
 `isspace`
 `printf`
 `strcpy`
 `tick_main_menu`

Parameters: none

Returns: none

2.7.9.2 fleas_atp_main_menu

The fleas_atp_main_menu function displays the Acceptance Test Menu supported by fleas_atp, and prompts the user for input. This function is called when the user selects the \$ ("DBCALIB locations") option from the Flea main menu. It is also called if the user enters ? at the "Flea ATP>" prompt.

The function call is fleas_atp_main_menu(). The function does the following:

- Clears the screen.
- Displays the menu.
- Sets the current prompt to "Flea ATP>".
- Displays the "Flea ATP>" prompt.

For a list of the options displayed on the Acceptance Test Menu, see fleas_atp.

Called By: fleas_atp
tick

Routines Called: blank
cup
printf
strcpy

Parameters: none

Returns: none

2.7.10 fleas_bal_opts.c

The functions in the fleas_bal_opts.c CSU let the Flea user interact with Ballistics. These functions are:

- fleas_bal_opts
- fleas_bal_opts_main_menu

2.7.10.1 fleas_bal_opts

The fleas_bal_opts function processes Ballistics requests input by the Flea user. This function is called via the *fleas_menu function pointer when the user selects the # ("ballistics") option from the Flea main menu.

The function call is `flea_bal_opts()`. `flea_bal_opts` prompts the user for a selection, then uses the `*flea_getc` function pointer to get the keystroke entered.

The following table lists the options supported by `flea_bal_opts`, and shows the steps it performs to process each one. This menu is displayed by `flea_bal_opts_main_menu`. Options flagged with an asterisk are supported by `flea_bal_opts` but do not appear on the menu.

Flea Ballistics Menu Option	Processing by <code>flea_bal_opts</code>
?* display menu	Calls <code>flea_bal_opts_main_menu</code> .
! fire round	Sets <code>p_flea_out->shoot_flag</code> to TRUE.
* cancel last round	Sets <code>p_flea_out->shoot_flag</code> to 0xff.
+ add vehicle	Makes sure other vehicle limit (70) has not been reached; prompts user for vehicle type, ASID, distance, x offset, and z offset; generates transformation matrix; places all data in <code>p_otherveh_state</code> ; increments <code>otherveh_count</code> .
- delete vehicle	Decrements <code>otherveh_count</code> .
@ toggle auto fire	If <code>auto_fire_flag</code> is FALSE: prompts user for rounds per minute; initializes <code>hits_per_minute</code> , <code>hit_count</code> , and <code>frame_count</code> to 0; sets <code>hit_count_flag</code> and <code>auto_fire_flag</code> to TRUE. If <code>auto_fire_flag</code> is TRUE: sets <code>hit_count_flag</code> and <code>auto_fire_flag</code> to FALSE.
1* +z rot	Increases <code>htrate</code> by .1.
2* 0z rot	Sets <code>htrate</code> to 0.00.
3* -z rot	Decreases <code>htrate</code> by .1.
4* +x rot	Increases <code>flea_x_rot</code> by .1.
5* 0x rot	Sets <code>flea_x_rot</code> to 0.0.
6* -x rot	Decreases <code>flea_x_rot</code> by .1.
7* +y rot	Increases <code>flea_y_rot</code> by .1.
8* 0y rot	Sets <code>flea_y_rot</code> to 0.0.
9* -y rot	Decreases <code>flea_x_rot</code> by .1.
A disable agl	Sets <code>p_flea_out->req_agl</code> to FALSE.
a request agl	Sets <code>p_flea_out->req_agl</code> to TRUE.

C chord control	<p>Displays current chord type and prompts for new value; sets p_flea_out->proc_chord_type; displays current tracer type and prompts for new value; sets p_flea_out->proc_chord_tracer_type; displays current chord mode and prompts for new value; sets p_flea_out->proc_chord_mode.</p> <p>If chord mode is CHORD_MODE_FINITE: displays current chord length and prompts for new value; sets p_flea_out->proc_chord_length.</p> <p>If chord mode is CHORD_MODE_INTERV, displays current chord end point and prompts for new values; sets p_flea_out->proc_chord_end.x, y, and z.</p> <p>Displays current explosion type and prompts for new value; sets p_flea_out->explosion_type.</p>
D toggle int comp	Sets p_flea_out->round_mode.
d remove static veh	Prompts user for vehicle type and id; puts data in p_flea_out; sets p_flea_out->rem_staticveh_flag to TRUE.
e burst effect	Prompts user for explosion type; sets p_flea_out->explosion_type.
F toggle shot report	<p>If p_flea_out->round_mode indicates shot reporting is on: changes p_flea_out->round_mode to disable shot reporting.</p> <p>If p_flea_out->round_mode indicates shot reporting is off: changes p_flea_out->round_mode to enable shot reporting; prompts user for target id, sets p_flea_out->target_id.</p>
g toggle agl ground follow	<p>If p_flea_out->agl_terrain_follow is FALSE: sets p_flea_out->req_agl to TRUE; prompts user for height above ground; sets p_flea_out->desired_agl; sets p_flea_out->agl_terrain_follow to TRUE.</p> <p>If p_flea_out->agl_terrain_follow is TRUE: sets it to FALSE.</p>
h* zero velocity	Sets speed to 0.0.
J delete traj. table	(not currently implemented)
j add new traj. table	(not currently implemented)
L toggle laser weapon	If p_flea_out->proc_chord_flag is non-zero, sets to FALSE; if 0, sets to 9; displays new status.
l fire laser weapon	Sets p_flea_out->proc_chord_flag to TRUE.
n* decrease velocity	Decreases speed by .2.
p buffer pointers	Displays pointers to p_flea_in, p_flea_out, and demo_ptr.

R round control	Displays current round type, tracer flag, and round mode; prompts for new values; sets p_flea_out->new_round_type, tracer_flag, and round_mode. If round mode is ROUND_MODE_SHOT_REPORT: prompts user for target; sets p_flea_out->target_id. Displays current tracer, round, and explosion types; prompts for new values; sets p_flea_out->tracer_type, round_type, and explosion_type.
r change round type	Prompts user for round type; sets p_flea_out->round_type.
S toggle tracer display	Sets p_flea_out->round_mode.
s drop static veh	Prompts user for vehicle type, id, and ASID; generates transformation matrix; puts all data in p_flea_out; sets p_flea_out->add_staticveh_flag to TRUE.
T tracer type	Prompts user for tracer type; sets p_flea_out->tracer_type.
t traj chord	Prompts user for chord id type, tracer flag, starting and ending points; sets values in p_flea_out->p_traj_chord; sets p_flea_out->traj_chord_flag to TRUE.
V add vehicleS	Resets otherveh_count to 0; prompts user for type, number of vehicles, maximum distance, and ASID flag. For each vehicle: sets x,y,z offsets; generates transformation matrix; puts data in p_otherveh_state; increments otherveh_count.
v delete vehicleS	Sets otherveh_count to 0.
x* exit	Sets *flea_menu function pointer to tick; calls tick_main_menu.
y* increase velocity	Increases speed by .2.

Called By: flea_io_task (through *flea_menu)

Routines Called:

- *flea_getc
- bcopy
- blank
- cup
- flea_bal_opts_main_menu
- gets
- GLOB
- isprint
- isspace
- printf
- sscanf
- tick_main_menu

Parameters: none

Returns: none

2.7.10.2 flea_bal_opts_main_menu

The `flea_bal_opts_main_menu` function displays the Flea Ballistics menu, which lets the user interface to Ballistics during a Flea exercise. This function is called if the user selects the # ("ballistics") option from the Flea main menu. It is also called if the user enters ? at the "Flea Ballistics>" prompt.

The function call is `flea_bal_opts_main_menu()`. The function does the following:

- Clears the screen.
- Calls `menu_header` to display the labels for the Flea simulation statistics.
- Displays the menu.
- Calls `update_menu_header` to display the current Flea simulation statistics.
- If an AGL value has been returned, displays it.
- If the `auto_fire_flag` is enabled, displays the number of rounds per minute.
- If the `proc_chord_flag` is enabled, displays "Laser weapon is ON."
- If the `shot_report_flag` is enabled, displays the round and target ids, and the status of the last shot ("NONEXISTENT TARGET," "FELL SHORT," "DYNAMIC VEHICLE," or "STATIC VEHICLE.")
- Displays the "FLEA_BAL_OPTS>" prompt.

The options on this menu are processed by `flea_bal_opts`. For a list of the options, refer to `flea_bal_opts`.

Called By: `flea_bal_opts`
`tick`

Routines Called: `blank`
`cup`
`menu_header`
`printf`
`update_menu_header`

Parameters: none

Returns: none

2.7.11 flea_db_traverse.c

The `flea_db_traverse` function lets the Flea user traverse the database in a set direction and at a set speed. This feature is invoked by selecting the T ("start/restart db trv") option from the Flea main menu. The user is prompted for the direction in which to travel, the vehicle speed in meters per frame, the traversal interval in meters, and the boundaries in which to travel (minimum and maximum x,y coordinates). `flea_db_traverse` is called every frame by `flea_update_pos`, whether or not database traversal has been initiated.

The function call is **flea_db_traverse()**.

The function examines the value of `p_flea_out->db_trv_state` to determine whether database traversal is off, in progress, or to be stopped. It then proceeds as follows:

- If `db_trv_state = 0` (traversal is off - initial condition):
 - No processing.
- If `db_trv_state = 1` (traversal is on - set by `flea_db_traverse` after traversal is initiated by user):
 - Checks to see if the current position is beyond the set limits; if so, sets `db_trv_state` to 5 (no traversal running) and initializes `skidx`, `skidy`, `skidz`, `htrate`, `flea_x_rot`, `flea_y_rot` and speed to 0.
 - If the location is within limits and `db_trv_direction = 0` (y):
 - * Increases `flea_vppos.y` by `db_trv_speed`.
 - * If the new `flea_vppos.y` is out of bounds, increases `flea_vppos.x` by `db_trv_interval`; increases `flea_vprot.z` by 180.0; changes sign of `db_trv_speed`; if `agl_terrain_follow` is enabled, increases `flea_vppos.z` by 2000.0.
 - If the location is within limits and `db_trv_direction = 1` (x):
 - * Increases `flea_vppos.x` by `db_trv_speed`.
 - * If the new `flea_vppos.x` is out of bounds, increases `flea_vppos.y` by `db_trv_interval`; increases `flea_vprot.z` by 180.0; changes sign of `db_trv_speed`; if `agl_terrain_follow` is enabled, increases `flea_vppos.x` by 2000.0.
- If `db_trv_state = 2` (halt traversal - set by tick if user selects "stop db trv"):
 - Stores vehicle's current position and orientation in `db_trv_pos.x`, `y`, and `z` and `db_trv_orient.x`, `y`, and `z`.
 - Initializes `skidx`, `skidy`, `skidz`, `htrate`, `flea_x_rot`, and `flea_y_rot` to 0.
 - Sets `db_trv_state` to 0 (traversal off).
- If `db_trv_state` is 3 (turn traversal on - set by tick if user selects "start/restart db trv") or 4 (resume traversal - set by tick if user selects "resume db trv"):
 - Restores vehicle's position and orientation.
 - Initializes `skidx`, `skidy`, `skidz`, `htrate`, `flea_x_rot`, `flea_y_rot`, and speed to 0.
 - Sets `db_trv_state` to 1 (traversal on).

Called By: `flea_update_pos`

Routines Called: none

Parameters: none

Returns: none

2.7.12 `flea_decode_data.c`

The `flea_decode_data` function returns decodes the messages returned from the CIG real-time software. These messages are the same as those that would be returned to the Simulation Host during a simulation exercise. They include Ballistics responses, local

terrain messages, and error messages. If a message requires processing, `flea_decode_data` puts the message or other required data in `p_flea_in`.

The function call is `flea_decode_data()`. The function does the following:

- Initializes the variables in `p_flea_in` that indicate whether certain types of data have been returned from the CIG (`terrain_returned`, `agl_returned`, `range_returned[i]`, and `hit_returned[i]`).
- Processes each message in the `flea_omsg` packet (see table below).
- Keeps track of the packet size by incrementing a counter by the size of each message as it is processed.
- Stops processing if a `MSG_END` or invalid message is detected, or if the calculated packet size exceeds the length specified in the packet header.

The following table identifies the messages processed by `flea_decode_data` and shows the major steps it performs to process each one. Note that several messages cause no processing other than incrementing the packet size.

Message	Processing by flea_decode_data
MSG_AGL	Sets p_flea_in->rtm_agl to altitude specified in message; sets p_flea_in->agl_returned to TRUE; increments packet size.
MSG_END	Sets packet_end_flag to TRUE, which causes the function to exit the loop.
MSG_HIT_RETURN	If flea_hit_count_flag is TRUE, increments flea_hit_count; generates a MSG_SHOW_EFFECT message to display effect (p_flea_out->explosion_type) at intersection point specified in MSG_HIT_RETURN message; increments packet size.
MSG_LASER_RETURN	Sets p_flea_in->range_returned[n] to TRUE; copies message data to p_flea_in->laser_return[n]; increments packet size.
MSG_LOCAL_TERRAIN	Increments packet size.
MSG_LT_PIECE	Puts each chunk of data into local_ter array; truncates data if too large; increments packet size.
MSG_MISS	Increments miss_count; increments packet size.
MSG_PASS_BACK	Increments packet size.
MSG_RETURN_POINT_INFO	Increments packet size.
MSG_SHOT_REPORT	Sets p_flea_in->shot_report_flag to TRUE; copies data from message to p_flea_in->shot_report; increments packet size.
MSG_SYS_ERROR	Increments packet size.
MSG_TF_HDR	If vehicle id is valid, puts data from message into p_flea_veh[] array for specified vehicle id; increments packet size.
MSG_TF_PT	If vehicle id and point number are valid, puts data from message into p_flea_veh[] array for specified vehicle id/feedback point; increments count of feedback points received; increments packet size.

Called By: flea

Routines Called: asm
bcopy
printf

Parameters: none

Returns: none

2.7.13 `flea_demo.c`

The `flea_demo` function calls autopilot to run the autopilot demo. This function is called every frame if a demonstration has been initiated (i.e., demonstration is not equal to 0)

The function call is `flea_demo()`. The function does the following:

- If demonstration = 1:
 - Calls autopilot to run the demo.
 - Increments the frame number sent to autopilot.
 - If the new frame number exceeds the total number of frames in the demo (this value is returned by autopilot):
 - * Calls autopilot with frame number 9999 (stop demo).
 - * Sets demonstration = 0 (no demo in progress).
 - * Resets the frame number to 0.
- If demonstration = 99, 98, 96: not currently implemented.
- Stores the vehicle's `vpos.z` as `last_z`.

Called By: `flea_update_pos`

Routines Called: `autopilot`

Parameters: none

Returns: none

2.7.14 `flea_draw_2d.c`

The `flea_draw_2d` function reads 2-D overlay data from a file specified by the user. It puts the data into a structure that is used by `update_2d` to pass the new overlay data to the GSP. This function is called if the Flea user selects the `f` ("draw 2d via file") option from the Flea Switches menu.

The function call is `flea_draw_2d()`. The function does the following:

- Prompts the user for the subsystem (backend) id: 0, 1, or 3 for both.
- Prompts the user for the name of the `pass_on` message file that contains the 2-D commands.
- Opens the specified file.
- Reads each line of the file into the `draw_2d_buffer` array.
- Closes the 2-D file.
- Sets `draw_2d_buffer[draw_index][0]` to 8888. This signals the end of the data for `update_2d`.
- Sets `command_2d[]` for the specified subsystem(s) to 10. This tells `update_2d` that the command to be processed is `DRAW_2D`.

Called By: flea_switches

Routines Called: cup
fclose
fgets
fopen
isdigit
isspace
printf
scanf
sscanf
strcmp

Parameters: none

Returns: none

2.7.15 flea_encode_data.c

The `flea_encode_data` function is responsible for calling all of the `upd_*` functions that generate the outgoing (Simulation Host-type) messages to be returned to the simulation software. This function is called every frame during a Flea exercise.

The function call is `flea_encode_data()`. The function calls all of the functions listed below under Routines Called, with the following exceptions:

- The function calls `dynamic_demo` only if demonstration is set to 2. This indicates that the user has initiated a moving model demo from the Flea Switches, Flea AGPT Locations, or Flea AGPT Switches menu.
- The function calls `update_agpt_2d` or `update_2d` only if a change affecting 2-D overlays has been specified for a given subsystem.
 - `update_agpt_2d` is called if `type_update_2d` is 1. This value is set by `set_command_2d` if the option was selected from the Flea AGPT Switches menu.
 - `update_2d` is called if `type_update_2d` is *not* 1, indicating that the option was selected from the regular Flea Switches menu.

Each `upd_*` function looks in `p_flea_out` to determine whether it has any data to process this frame.

Called By: flea

Routines Called: dynamic_demo
send_ammo_define
send_gun_overlay
upd_add_static_veh
upd_auto_fire
upd_chord_fired

```

upd_clouds
upd_count_hits_per_min
upd_dynamic_matrix
upd_flea_vehicles
upd_lt_state
upd_matrix_values
upd_ppm
upd_rem_static_veh
upd_req_agl
upd_req_lrange
upd_req_point
upd_rotation_values
upd_round_fired
upd_send_dynamic
upd_send_stop
upd_show_eff
upd_sio_write
upd_subsys_mode
upd_view_flags
upd_view_mag
upd_view_mode
upd_viewport_up
update_2d
update_agpt_2d

```

Parameters: none

Returns: none

2.7.16 flea_graphics_test.c

The functions in the flea_graphics_test.c CSU are used to run acceptance tests. These functions are:

- flea_graphics_test
- flea_graphics_test_main_menu

2.7.16.1 flea_graphics_test

The flea_graphics_test function processes the user's selection on the Flea Graphics Test Menu. The options on this menu are used to run acceptance tests. This menu becomes active (via the *flea_menu function pointer) if the user selects the & ("DBTEST locations") option from the Flea main menu.

The function call is **flea_graphics_test()**. The function uses the *flea_getc function pointer to get the keystroke input by the user, then processes it.

The following table identifies the options supported by flea_graphics_test, and shows the steps it performs to process each one. For each test, the function sets the simulated vehicle's position to the location in that database that contains the selected test pattern.

The Graphics Test menu is displayed by `flea_graphics_test_main_menu`. Options supported but not shown on the menu are flagged below with an asterisk.

Graphics Test Menu Option	Processing by <code>flea_graphics_test</code>
? help	No action.
0 string of vehicles	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
1 striped right triangle	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
2 44x44 chessboard	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
3 32x32 chessboard	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
4 house	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
5 4x4 chessboard (textures)	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
6 shaded textures	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
7 66 sided pyramid	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
8 4 tanks (no textures)	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
9 tank behind explosion	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
a tank behind treeline	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
b tank behind tree	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
c 2 pixel vertical line	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
d 50 cubes	Sets <code>flea_vppos</code> and <code>flea_vprot</code> .
q* quit	Sets <code>*flea_menu</code> function pointer to tick; sets prompt to "Flea>"; calls <code>tick_main_menu</code> .
x* exit	Sets <code>*flea_menu</code> function pointer to tick; sets prompt to "Flea>"; calls <code>tick_main_menu</code> .

After each selection is processed, `flea_graphics_test` initializes `skidx`, `skidy`, `skidz`, `htrate`, and `speed` to 0.

Called By: `flea_io_task` (through `*flea_menu`)

Routines Called: `*flea_getc`
`cup`
`GLOB`
`isprint`
`isspace`
`printf`
`tick_main_menu`
`strcpy`

Parameters: none

Returns: none

2.7.16.2 flea_graphics_test_main_menu

The `flea_graphics_test_main_menu` function displays the Flea Graphics Test menu. This function is called if the user selects the & ("DBTEST locations") from the Flea main menu.

The function call is `flea_graphics_test_main_menu()`. The function does the following:

- Clears the screen.
- Displays the menu.
- Displays the "Flea Graphics>" prompt.

The option selected by the user is processed by `flea_graphics_test`. For a list of the options displayed on the menu, refer to `flea_graphics_test`.

Called By: tick

Routines Called: blank
cup
printf
strcpy

Parameters: none

Returns: none

2.7.17 flea_init_cig_sw.c

The `flea_init_cig_sw` function opens the Flea configuration file, calls `config_translator` to process its entries, then builds a message packet to return to the CIG. This function is called when fleas are initialized.

The Flea configuration file can be specified using the `c` ("change configfile name") option on the Gossip main menu. If no file was specified, `flea_init_cig_sw` looks for a default file.

The function call is `flea_init_cig_sw()`. The function does the following:

- Initializes various variables that pertain to the message packet.
- If a configuration file name was entered through Gossip, calls `find_fn` to open the specified file; tries first for an exact match, then for a partial match.
- If no file was specified through Gossip, calls `find_fn` to open the highest-numbered `cfg*` file.
- Allocates memory for the structure used to build the CIG configuration messages.
- Calls `config_translator` to read the file and generate the CIG configuration messages.

- Processes each message built by the config_translator functions (see table below).
- Stops processing messages when a MSG_CIG_CTL or MSG_END message is detected, or if the packet count exceeds the maximum packet size.
- Uses the EXCHANGE_FLEA_DATA macro (described in Appendix B) to exchange message packets with the real-time software.
- Resets the packet_count to 0 and the packet_complete_flag to FALSE.
- Frees the memory allocated to process the messages.
- Uses the EXCHANGE_FLEA_DATA macro 16 more times, to ensure that the CIG has fully completed the configuration stage.

The following table lists the message types that may be generated by the config_translator functions, and identifies the major steps flea_init_cig_sw uses to process them. In most cases, flea_init_cig_sw simply uses the OUTPUT_MESSAGE macro (described in Appendix B) to add the message to the outgoing message packet (flea_imsig). The function also keeps track of the size of the packet and, if FLEA_DEBUG is enabled, outputs the contents of many of the messages to stdout.

Message Type	Processing by flea_init_cig_sw
MSG_2D_SETUP	Uses OUTPUT_MESSAGE to add message to packet.
MSG_ADD_TRAJ_TABLE	Uses OUTPUT_MESSAGE to add message to packet.
MSG_AMMO_DEFINE	Uses OUTPUT_MESSAGE to add message to packet.
MSG_CIG_CTL	Copies message to packet; sets packet_complete_flag to TRUE (which causes flea_init_cig_sw to exit loop).
MSG_CREATE_CONFIGNODE	Copies message to packet; if node_index = 1 (initial world-to-hull matrix), calls id_4x3mtx, rotate_x_nt, rotate_y_nt, rotate_z_nt, and translate to generate transformation matrix, and adds matrix to message.
MSG_DEFINE_TX_MODE	Uses OUTPUT_MESSAGE to add message to packet.
MSG_DR11_PKT_SIZE	Uses OUTPUT_MESSAGE to add message to packet.
MSG_END	Sets completion_flag to TRUE; sets packet_complete_flag to TRUE (which causes flea_init_cig_sw to exit loop).
MSG_FILE_DESCR	Copies message to packet. If flea_color_cfg_wanted_G is enabled: sets db_req to DB_COLOR_CFG_0 and db_name to color_cfg_fn_to_use. If flea_esifa_load_wanted_G is enabled: sets db_req to DB_ESIFA_LOAD_0 and db_name to esifa_fn_to_use.
MSG_LT_STATE	Uses OUTPUT_MESSAGE to add message to packet.
MSG_OVERLAY_SETUP	Uses OUTPUT_MESSAGE to add message to packet.
MSG_PPM_DISPLAY_MODE	Uses OUTPUT_MESSAGE to add message to packet.
MSG_PPM_DISPLAY_OFFSET	Uses OUTPUT_MESSAGE to add message to packet.
MSG_PPM_PIXEL_LOCATION	Uses OUTPUT_MESSAGE to add message to packet.
MSG_PPM_PIXEL_STATE	Uses OUTPUT_MESSAGE to add message to packet.
MSG_SIO_CLOSE	Uses OUTPUT_MESSAGE to add message to packet.
MSG_SIO_INIT	Uses OUTPUT_MESSAGE to add message to packet.
MSG_TF_INIT_HDR	Uses OUTPUT_MESSAGE to add message to packet.
MSG_TF_INIT_PT	Uses OUTPUT_MESSAGE to add message to packet.
MSG_TF_STATE	Uses OUTPUT_MESSAGE to add message to packet.
MSG_TRAJ_ENTRY	Uses OUTPUT_MESSAGE to add message to packet.
MSG_VIEWPORT_STATE	Uses OUTPUT_MESSAGE to add message to packet.

If an invalid message type is detected, flea_init_cig_sw displays an error message and exits with a 1.

The function returns 1 if successful. It returns EOF if it could not find the file specified through Gossip, or if no cfg* file was found on disk. The function exits with a 1 if it could not allocate enough memory to build messages, or if config_translator returns an error.

Called By: flea

Routines Called: bcopy
 config_translator
 cos
 EXCHANGE_FLEA_DATA
 exit
 find_fn
 free
 GLOB
 id_4x3mtx
 malloc
 OUTPUT_MESSAGE
 printf
 rotate_x_nt
 rotate_y_nt
 rotate_z_nt
 sin
 strcpy
 strlen
 TORAD
 translate

Parameters: none

Returns: EOF
 1

2.7.18 **flea_ppm_obj.c**

The functions in the flea_ppm_obj.c CSU control PPM parameter downloads to the CIG. These functions are:

- upd_ppm
- flea_calibration_image
- flea_ppm_display_mode
- flea_ppm_display_offset
- flea_ppm_pixel_location
- flea_ppm_pixel_state

2.7.18.1 **upd_ppm**

The upd_ppm function checks p_flea_out for any PPM requests. If it finds any, it calls the appropriate routine to add the message to the outgoing message packet. This function is called at the end of every frame during a Flea exercise.

upd_ppm determines whether it has any data to process by examining various flags in p_flea_out. These flags are set if the user selects various options from the Flea PPM menu, which is reached by selecting P ("Calibrate PPMs") from the Flea main menu. These flags and the options used to set them are as follows:

p_flea_out Flag	Flea PPM Menu Option
calibration_image_flag	i ("cal image")
ppm_display_mode_flag	m ("change mode")
ppm_display_offset_flag	1 ("display left"), 2 ("display down"), 3 ("display right"), 5 ("display up")
ppm_pixel_location_flag	h ("pixel left"), l ("pixel right"), j ("pixel down"), k ("pixel up")
ppm_pixel_state_flag	o ("pixel off"), O ("pixel on")
auto_ppm_flag	a ("auto test")

The function call is **upd_ppm()**. The function does the following:

- If calibration_image_flag in p_flea_out is set:
 - Makes sure the packet has enough room for the new message.
 - Increments the packet size.
 - Calls flea_calibration_image to build the MSG_CALIBRATION_IMAGE message and put it in the outgoing packet.
 - Sets calibration_image_flag to FALSE.
- If ppm_display_mode_flag in p_flea_out is set:
 - Makes sure the packet has enough room for the new message.
 - Increments the packet size.
 - Calls flea_ppm_display_mode to build the MSG_PPM_DISPLAY_MODE message and put it in the outgoing packet.
 - Sets ppm_display_mode_flag to FALSE.
- If ppm_display_offset_flag in p_flea_out is set:
 - Makes sure the packet has enough room for the new message.
 - Increments the packet size.
 - Calls flea_ppm_display_offset to build the MSG_PPM_DISPLAY_OFFSET message and put it in the outgoing packet.
 - Sets ppm_display_offset_flag to FALSE.
- If ppm_pixel_location_flag in p_flea_out is set:
 - Makes sure the packet has enough room for the new message.
 - Increments the packet size.
 - Calls flea_ppm_pixel_location to build the MSG_PPM_PIXEL_LOCATION message and put it in the outgoing packet.
 - Sets ppm_pixel_location_flag to FALSE.
- If ppm_pixel_state_flag in p_flea_out is set:
 - Makes sure the packet has enough room for the new message.
 - Increments the packet size.
 - Calls flea_ppm_pixel_state to build the MSG_PPM_PIXEL_STATE message and put it in the outgoing packet.
 - Sets ppm_pixel_state_flag to FALSE.
- If auto_ppm_flag in p_flea_out is set:
 - Increments ppm_frame_count.
 - Determines which period (if any) is up this frame. (The user specifies the periods for each PPM operation when choosing the "auto test" function.)
 - * If display_mode_period: sets display_mode_chan to 0 and display_mode_flag to TRUE.

- * If display_offset_period: sets display_offset_chan to 0 and display_offset_flag to TRUE.
- * If cal_pixel_period: sets cal_pixel_chan to 0 and cal_pixel_flag to TRUE.
- * If pixel_loc_period: sets pixel_loc_chan to 0 and pixel_loc_flag to TRUE.
- Processes the change based on which flag is now set:
 - * If display_mode_flag: sets all data in msg_ppm_display_mode; makes sure packet has enough room; calls flea_ppm_display_mode to put MSG_PPM_DISPLAY_MODE message in packet; increments display_mode_chan; if channel is 8 or more, resets display_mode, display_mode_chan, and display_mode_flag.
 - * If display_offset_flag: sets all data in msg_ppm_display_offset; makes sure packet has enough room; calls flea_ppm_display_offset to put MSG_PPM_DISPLAY_OFFSET message in packet; increments display_offset_chan; if channel is 8 or more, resets display_offset_i and j, display_offset_chan, and display_offset_flag.
 - * If cal_pixel_flag: sets all data in msg_ppm_pixel_state; makes sure packet has enough room; calls flea_ppm_pixel_state to put MSG_PPM_PIXEL_STATE message in packet; increments cal_pixel_chan; if channel is 4 or more, resets pixel_on_off, cal_pixel_chan, and cal_pixel_flag.
 - * If pixel_loc_flag: sets all data in msg_ppm_pixel_location; makes sure packet has enough room; calls flea_ppm_pixel_location to put MSG_PPM_PIXEL_STATE message in packet; increments pixel_loc_chan; if channel is 4 or more, resets pixel_loc_i and j, pixel_loc_chan, and pixel_loc_flag.

The function calls flea_error_printf to output an error if the packet does not have enough room for the new message.

Called By: flea_encode_data

Routines Called: flea_calibration_image
flea_error_print
flea_ppm_display_mode
flea_ppm_display_offset
flea_ppm_pixel_location
flea_ppm_pixel_state

Parameters: none

Returns: none

2.7.18.2 flea_calibration_image

The flea_ppm_pixel_state function puts the MSG_CALIBRATION_IMAGE message into the outgoing message packet. This function is called if the calibration_image_flag in

`p_flea_out` is set to TRUE because the user selected the `i` ("cal image") option from the Flea PPM menu.

The function call is `flea_calibration_image(p_msg, p_new_msg)`, where:

`p_msg` is a pointer to the outgoing message packet (`flea_img`)

`p_new_msg` is a pointer to the message in `p_flea_out`

After adding the message and its header, the function increments the pointer to the outgoing packet and returns it as `p_msg`.

Called By: `upd_ppm`

Routines Called: `none`

Parameters: `MSG_CALIBRATION_IMAGE` `*p_msg`
 `MSG_CALIBRATION_IMAGE` `*p_new_msg`

Returns: `p_msg`

2.7.18.3 `flea_ppm_display_mode`

The `flea_ppm_display_mode` function puts the `MSG_PPM_DISPLAY_MODE` message into the outgoing message packet. This function is called if the `ppm_display_mode_flag` in `p_flea_out` is set to TRUE because the user selected the `m` ("change mode") option from the Flea PPM menu. It is also called if the `a` ("auto test") option was selected and the `display_mode_period` expires this frame.

The function call is `flea_ppm_display_mode(p_msg, p_new_msg)`, where:

`p_msg` is a pointer to the outgoing message packet (`flea_img`)

`p_new_msg` is a pointer to the message in `p_flea_out`

After adding the message and its header, the function increments the pointer to the outgoing packet and returns it as `p_msg`.

Called By: `upd_ppm`

Routines Called: `none`

Parameters: `MSG_PPM_DISPLAY_MODE` `*p_msg`
 `MSG_PPM_DISPLAY_MODE` `*p_new_msg`

Returns: `p_msg`

2.7.18.4 flea_ppm_display_offset

The `flea_ppm_display_offset` function puts the `MSG_PPM_DISPLAY_OFFSET` message into the outgoing message packet. This function is called if the `ppm_display_offset_flag` in `p_flea_out` is set to `TRUE` because the user selected any of the "display <direction>" options from the Flea PPM menu. It is also called if the `a` ("auto test") option was selected and the `display_offset_period` expires this frame.

The function call is `flea_ppm_display_offset(p_msg, p_new_msg)`, where:

p_msg is a pointer to the outgoing message packet (`flea_ims`)

p_new_msg is a pointer to the message in `p_flea_out`

After adding the message and its header, the function increments the pointer to the outgoing packet and returns it as *p_msg*.

Called By: `upd_ppm`

Routines Called: `none`

Parameters: `MSG_PPM_DISPLAY_OFFSET` **p_msg*
 `MSG_PPM_DISPLAY_OFFSET` **p_new_msg*

Returns: `p_msg`

2.7.18.5 flea_ppm_pixel_location

The `flea_ppm_pixel_location` function puts the `MSG_PPM_PIXEL_LOCATION` message into the outgoing message packet. This function is called if the `ppm_pixel_location_flag` in `p_flea_out` is set to `TRUE` because the user selected any of the "pixel <direction>" options from the Flea PPM menu. It is also called if the `a` ("auto test") option was selected and the `pixel_loc_period` expires this frame.

The function call is `flea_ppm_pixel_location(p_msg, p_new_msg)`, where:

p_msg is a pointer to the outgoing message packet (`flea_ims`)

p_new_msg is a pointer to the message in `p_flea_out`

After adding the message and its header, the function increments the pointer to the outgoing packet and returns it as *p_msg*.

Called By: `upd_ppm`

Routines Called: `none`

Parameters: MSG_PPM_PIXEL_LOCATION *p_msg
MSG_PPM_PIXEL_LOCATION *p_new_msg

Returns: p_msg

2.7.18.6 flea_ppm_pixel_state

The `flea_ppm_pixel_state` function puts the MSG_PPM_PIXEL_STATE message into the outgoing message packet. This function is called if the `ppm_pixel_state_flag` in `p_flea_out` is set to TRUE because the user selected o ("pixel off") or O ("pixel on") from the Flea PPM menu. It is also called if the a ("auto test") option was selected and the `cal_pixel_period` expires this frame.

The function call is `flea_ppm_pixel_state(p_msg, p_new_msg)`, where:

p_msg is a pointer to the outgoing message packet (`flea_imsig`)

p_new_msg is a pointer to the message in `p_flea_out`

After adding the message and its header, the function increments the pointer to the outgoing packet and returns it as *p_msg*.

Called By: upd_ppm

Routines Called: none

Parameters: MSG_PPM_PIXEL_STATE *p_msg
MSG_PPM_PIXEL_STATE *p_new_msg

Returns: p_msg

2.7.19 flea_script.c

The functions in the `flea_script.c` CSU provide the ability to play back a recorded script file. These functions are:

- `flea_abs_playback`
- `get_next_packet`
- `get_next_message`

A script file can be built by selecting the r ("record i/f messages menu") option from the Gossip main menu. If this feature is enabled, the `cigsimio` functions in the Real-Time Processing CSC save all incoming and outgoing messages to a binary file. Recording continues for a specified number of frames.

2.7.19.1 **flea_abs_playback**

The **flea_abs_playback** function plays back the script recorded in a file. This function is called every frame by **flea** (instead of the functions called when a user is driving the exercise) if the **fleaG_abs_playback** variable is TRUE. This variable is enabled if the user selects the **b** ("begin absolute playback") option from the Flea Script menu. The user must also enter the name of the file that contains the script to be played back.

When the recorded file is played back, **flea_abs_playback** extracts all of the incoming (SIM-to-CIG) messages from the file and places them in the **flea_imsig** packet. Packets are then exchanged as usual by the **flea** function.

The function call is **flea_abs_playback()**. The function does the following:

- If a playback file is not yet open, opens it in read-only mode.
 - If the file cannot be opened, displays an error and sets **fleaG_abs_playback** to FALSE.
 - If the file is opened successfully, sets **playback_file_open** to TRUE.
- If **fleaG_stop_playback** is TRUE (because the user selected the "stop playback" option from the Flea Script menu):
 - Sets **playback_file_open** to FALSE.
 - Closes the playback file.
 - Sets **fleaG_abs_playback** to FALSE.
 - Sets **fleaG_stop_playback** to FALSE.
 - Exits.
- Sets **end_of_frame** to FALSE.
- Calls **get_next_packet** to get the next message packet.
 - If **get_next_packet** reports an end-of-file, outputs a message and rewinds the file.
 - If the packet is tagged **TAG_MSG_FROM_CIG** or **TAG_PKT_FROM_CIG**: sets **end_of_frame** to TRUE.
 - If the packet is tagged **TAG_PKT_TO_CIG**: copies the packet to **flea_imsig** and sets **end_of_frame** to TRUE.
 - If the packet is tagged **TAG_MSG_TO_CIG**: calls **get_next_message** to get the message, then copies the message to **flea_imsig**.
- When the end of frame is reached, adds a **MSG_END** message to the **flea_imsig** packet.

Called By: **flea**

Routines Called: **bcopy**
get_next_message
get_next_packet
ifx_close
ifx_open
ifx_sposn
printf

Parameters: **none**

Returns: none

2.7.19.2 get_next_packet

The `get_next_packet` function gets the next message packet from the script file.

The function call is `get_next_packet(fd, pkt_code, pkt_addr, pkt_size)`, where:

fd is the file descriptor of the record (script) file

pkt_code is a pointer to the packet's tag (TAG_STRING, TAG_PKT_TO_CIG, TAG_MSG_TO_CIG, TAG_PKT_FROM_CIG, or TAG_MSG_FROM_CIG.)

pkt_addr is a pointer to the packet found by `get_next_packet`

pkt_size is a pointer to the size of the packet found by `get_next_packet`

The function does the following:

- Reads the packet tag.
- Sets **pkt_code*, **pkt_addr*, and **pkt_size*.
- Reads the packet buffer.
- Verifies that the number of bytes read equals the block size specified in the packet tag.

The function returns the status returned by `ifx_read` if successful. It returns EOF if it encounters an error reading the script file.

Called By: `flea_abs_playback`

Routines Called: `ifx_read`
`printf`

Parameters:	INT_4	fd
	UNS_1	*pkt_code
	UNS_1	**pkt_addr
	UNS_2	*pkt_size

Returns: EOF
status

2.7.19.3 get_next_message

The `get_next_message` function gets the next message from the script file. This function is called if the entry in the script file is tagged TAG_MSG_TO_CIG.

The function call is `get_next_message(msg_hdr_P, msg_code_P, data_size_P)`, where:

msg_hdr_P is a pointer to the message header
msg_code_P is a pointer to the message code
data_size_P is a pointer to the message size

The function does the following:

- Sets **msg_code_P* to the message type specified in the message header.
- Sets **data_size_P* to the size of the message header plus the message length specified in the header.
- Validates the message code.

The function returns 0 if successful. It returns EOF if the message code is invalid.

Called By: `flea_abs_playback`

Routines Called: `printf`

Parameters:	<code>MSG_HDR</code>	<code>*msg_hdr_P</code>
	<code>INT_2</code>	<code>*msg_code_P</code>
	<code>INT_2</code>	<code>*data_size_P</code>

Returns: `EOF`
`0`

2.7.20 `flea_simulate_vehicles.c`

The functions in the `flea_simulate_vehicles.c` CSU are used to update the position and orientation values of each Flea vehicle based on terrain feedback. These functions are:

- `flea_simulate_vehicles`
- `find_pitch_and_roll`

2.7.20.1 `flea_simulate_vehicles`

The `flea_simulate_vehicles` function adjusts each Flea vehicle's position and orientation based on the terrain feedback points received. This function is called every frame during a Flea exercise.

The `p_flea_veh[]` array (data type `SIM_VEH_STRUCT`) is used to maintain data for all Flea vehicles. Each vehicle's entry specifies its type, vehicle id, ASID value, transformation matrix, desired altitude, terrain feedback points, etc. The first vehicle in the array (index 0) is the simulated vehicle. The maximum number of Flea vehicles is specified by `MAX_FLEA_VEHICLES`, which is set in the `demo_struct.h` file.

The function call is `flea_simulate_vehicles()`. The function does the following:

- For the simulated vehicle:

- Sets a pointer to the vehicle's data in `p_flea_veh`. The simulated vehicle is the first element in the array.
- Makes sure the vehicle's in-use flag is TRUE and that all of its terrain feedback points have been received.
- Resets the received point count to 0.
- Sets the vehicle's `pos.x` to `flea_vppos.x`.
- Sets the vehicle's `pos.y` to `flea_vppos.y`.
- Sets the vehicle's `pos.z` and `flea_vppos.z` to the sum of (1) the average height of the vehicle's first three terrain feedback points, and (2) the vehicle's desired altitude.
- Adds the `z` values of all of the vehicle's terrain feedback points.
- Sets `flea_vppos.z` to the vehicle's desired altitude plus the average height of the feedback points.
- Sets speed to the vehicle's `dpos.y`.
- Increments the vehicle's `rot.z` by `drot.z`.
- Uses the CHECKROT macro to validate the value of `rot.z`.
- Sets `flea_vprot.z` to the vehicle's `rot.z`.
- Calls `find_pitch_and_roll` to calculate the vehicle's `rot.x` (pitch) and `rot.y` (roll) values.
- Sets `flea_vprot.x` to the vehicle's `rot.x`.
- Sets `flea_vprot.y` to the vehicle's `rot.y`.
- For all other vehicles, to the maximum specified by MAX_FLEA_VEHICLES:
 - Sets a pointer to the vehicle's data in `p_flea_veh`.
 - Makes sure the vehicle's in-use flag is TRUE and that all of its terrain feedback points have been received.
 - Resets the received point count to 0.
 - Increments the vehicle's `pos.x` by `dpos.y` times `mtx[1][0]`.
 - Increments the vehicle's `pos.y` by `dpos.y` times `mtx[1][1]`.
 - Increments the vehicle's `desired_alt` by `dpos.z`.
 - Adds the `position.z` values for all of the vehicle's feedback points.
 - Sets the vehicle's `pos.z` to the desired altitude plus the average height of the terrain feedback points.
 - Increments `rot.z` by `drot.z`.
 - Uses the CHECKROT macro to validate the `rot.z` value.
 - Calculates the cosine and sine of the vehicle's `x`, `y`, and `z` rotation values.
 - Calls `id_4x3mtx` to create an identity matrix.
 - Calls `rotate_z_nt` to rotate the identity matrix using the vehicle's `z` rotation value.
 - Calls `find_pitch_and_roll` to calculate the vehicle's `rot.x` (pitch) and `rot.y` (roll) values.
 - Calls `make4x3` to generate the vehicle's new transformation matrix.
 - Puts the new matrix and vectors into the vehicle's structure.
 - If the vehicle's `shoot_flag` is TRUE:
 - * Sets the `shoot_flag` to FALSE.
 - * Generates a MSG_PROCESS_ROUND message and puts it into the `flea_imsg` packet.
 - If the vehicle's `round_limit` flag is TRUE (auto-firing) and the `round_counter` exceeds the `round_limit`:
 - * Resets the `round_counter` to 0.
 - * Generates a MSG_PROCESS_ROUND message and puts it into the `flea_imsg` packet.

Called By: `flea_update_pos`

Routines Called: CHECKROT
 cos
 find_pitch_and_roll
 id_4x3mtx
 make4x3
 rotate_z_nt
 sin
 TORAD

Parameters: none

Returns: none

2.7.20.2 find_pitch_and_roll

The find_pitch_and_roll function calculates a Flea vehicle's pitch (rot.x) and roll (rot.y), based on the vehicle's three terrain feedback points. This function is called when flea_simulate_vehicles is updating the orientation of each Flea vehicle.

The function call is **find_pitch_and_roll(veh_P)**, where *veh_P* is a pointer to the vehicle's data in the *p_flea_veh[]* array. The function does the following:

- Computes the vector connecting the first and second terrain feedback points.
- Computes the vector connecting the second and third terrain feedback points.
- Computes the normal vector using the first two vectors.
- Calls *vec_mat_mul* to multiply the normal vector by the vehicle's matrix.
- Computes the magnitude of the normal vector (*mag_normal*) by taking the square root of ($\text{vec.x}^2 + \text{vec.y}^2 + \text{vec.z}^2$).
- Calculates the vehicle's pitch in radians as $[\pi / 2 - \text{acos}((\text{vec.y} / \text{mag_normal}))]$.
- Calculates the vehicle's roll in radians as $[\text{acos}((\text{vec.x} / \text{mag_normal})) - \pi / 2]$.
- Sets the vehicle's rot.x value to the value of pitch converted to degrees (using the *TODEG* macro).
- Sets the vehicle's rot.y value to the value of roll converted to degrees (using *TODEG*).

Called By: flea_simulate_vehicles

Routines Called: acos
 sqrt
 TODEG
 vec_mat_mul

Parameters: SIM_VEH_STRUCT *veh_P

Returns: none

2.7.21 flea_switches.c

The functions in flea_switches.c CSU let the user modify various simulation parameters. These functions are:

- flea_switches
- flea_switches_main_menu

2.7.21.1 flea_switches

The flea_switches function process the user's selection on the Flea Switches menu. This function is called (through the *flea_menu function pointer) if the user selects the * ("switches") option from the Flea main menu. The Flea Switches menu is used to:

- Modify simulation variables such as viewport parameters, system view flags, branch values, cloud models, and gun overlays.
- Initiate requests for AGL, laser range, local terrain, and terrain feedback processing.
- Start a moving model demo, or access the menu used to change demonstration parameters.

The function call is **flea_switches()**. flea_switches uses the *flea_getc function pointer to get the keystroke entered by the user, then processes the selection. In most instances, flea_switches places the values entered by the user into p_flea_out and sets the applicable flag in p_flea_out to trigger processing by the upd_* functions in encode_routines.c.

The following table lists the options supported by flea_switches, and shows the steps it takes to process each one. The menu is displayed by flea_switches_main_menu. Options flagged with an asterisk are supported by flea_switches but do not appear on the menu.

Flea Switches Menu Option	Processing by flea_switches
?* display menu	Calls flea_switches_main_menu.
! shoot gun	Sets p_flea_out->shoot_flag to TRUE.
1 1st text line	Clears screen; prompts user for text string (maximum 31 characters); puts data in text1_2d[].
2 2d mode	Clears screen; displays valid commands; prompts user for subsystem id and command to send; validates subsystem id. If specified subsystem contains a Force board: sets subsystem's element in p_flea_out->sub_2d_io[] to TRUE; puts command in subsystem's element in command_2d[]; sets subsystem's element in gsp_io_flag[] to TRUE.
3 2nd text line	Clears screen; prompts user for text string (maximum 31 characters); puts data in text2_2d[].
4 change lt state	Prompts user for new state (0 or 1), size, and frame interval; sets lt_state_code, lt_state_size, lt_state_interval; sets lt_state_flag to TRUE.
5 magnification	Prompts user for node index, lod multiplier, fov (i,j); puts values in p_flea_out->view_magnification; sets p_flea_out->view_mag_flag to TRUE.
6 subsys mode	Clears screen; displays color selection table; prompts user for subsystem id, color table, fade value, and special use mask; puts new values in p_flea_out; sets p_flea_out->subsys_mode_flag to TRUE.
7 viewport upd	Prompts user for viewport id, on/off, alternate on/off, modifier on/off; puts values in p_flea_out->vpt_update; sets p_flea_out->vpt_update_flag to TRUE.
8 view local terrain	Clears screen. If single-step mode is off: prompts user to set it. If local_terrain_wanted is TRUE: displays frame number and all data from local terrain message (number of polygons and bvols, all polygon entries, all bvol entries). If local_terrain_wanted is FALSE: displays error.
9 ammo define msg	Prompts user for ammo define map; puts values in flea_ammo_define.ammo_type_map; sets flea_ammo_define_flag to TRUE.
a request agl	Sets p_flea_out->req_agl to TRUE.
b branch value	Clears screen; displays current branch values; asks if user wants to change; if yes, prompts for index and new value; sets p_flea_out->branch_value[index].
C clouds message	Clears screen; prompts user for cloud state, cloud top altitude and model id, cloud bottom altitude and model id; puts values in p_flea_out; sets p_flea_out->cloud_flag to TRUE; displays "Command>" prompt.
c change cal modifier	Prompts user for new value; sets cal_modifier in global memory.

d moving model demo	If demonstration is 0 (model demo not running): initializes flea_vprot.x, flea_vprot.y, flea_vprot.z, skidx, skidy, skidz, htrate, and speed to 0; calls dynamic_demo with frame number set to 0; prompts user to select 'd' option again to adjust options. If demonstration is 2 (model demo already in progress): calls model_demo.
e transform update	Clears screen; prompts user for node index and type of rotation (ROT2x1, 1ROTATION, or 3ROTATIONS); sets rotation_type[node_index]; prompts user for rotation axis; sets rotation_axis[node_index]; prompts user for rotation in degrees per frame; sets rotation_incr[node_index].
f draw 2d via file	Calls flea_draw_2d.
g gun barrel update	Clears screen; prompts user for vehicle type, ready, malfunction indicator, multiple return indicator, ammo type, rangedg4 digits, azimuth digits, range2 digits, rotation of gun barrel; sets new gn_* values; sets send_gun_stat to TRUE.
l request laser range	Clears screen; prompts user for subsystem id, request id, channel number, pixel location (i,j); puts values in p_flea_out; sets p_flea_out->req_lrange to TRUE.
R request point info	Clears screen; prompts user for x and y coordinates of point; sets p_flea_out->request_point.x, y; sets p_flea_out->req_point to TRUE.
t transform values	(not currently implemented)
v change view flags	Prompts user for new view flags; sets p_flea_out->view_flag; displays "Command>" prompt.
x* exit	Sets *flea_menu function pointer to tick; sets current prompt to "Flea>"; calls tick_main_menu.

Called By: flea_io_task (through *flea_menu)

Routines Called:

- *flea_getc
- blank
- cup
- dynamic_demo
- flea_draw_2d
- flea_switches_main_menu
- GLOB
- isprint
- isspace
- model_demo
- printf
- scanf
- strcpy
- tick_main_menu

Parameters: none

Returns: none

2.7.21.2 flea_switches_main_menu

The `flea_switches_main_menu` function displays the Flea Switches menu. It also displays the latest data returned from the real-time software for AGL processing and laser range requests. `flea_switches_main_menu` is called if the user selects the * ("switches") option from the Flea main menu.

The function call is `flea_switches_main_menu()`. The function does the following:

- Clears the screen.
- Calls `menu_header` to display the field labels for the Flea simulation vehicle statistics displayed at the top of the screen.
- Displays the color description and fade value for each backend.
- Displays the Flea Switches menu.
- Calls `update_menu_header` to display current statistics on the Flea simulation vehicle.
- If an AGL value has been returned from the CIG (i.e., `p_flea_in->agl_returned` is TRUE), displays the value.
- If a laser range value has been returned from the CIG for subsystem 0 (i.e., `p_flea_in->range_returned[0]` is TRUE), displays the backend id, channel number, and range value.
- If a laser range value has been returned from the CIG for subsystem 1 (i.e., `p_flea_in->range_returned[1]` is TRUE), displays the backend id, channel number, and range value.
- Copies "Flea Switches>" to the current prompt.
- Displays the current prompt.

The menu option selected by the user is processed by `flea_switches`. For a list of the options displayed on this menu, see `flea_switches`.

Called By: `flea_switches`
`tick`

Routines Called: `blank`
`cup`
`menu_header`
`printf`
`strcpy`
`update_menu_header`

Parameters: none

Returns: none

2.7.22 `flea_update_pos.c`

The `flea_update_pos` function is responsible for moving and rotating the simulated vehicle each frame. The function first calls various functions to move the vehicle if any type of automated movement has been requested (e.g., database traversal, AGL terrain follow, or a moving demonstration). It then generates the vehicle's new transformation matrix based on the position and rotation values calculated by the automatic movement functions or by the user's entries on various menus.

The function call is `flea_update_pos()`. The function does the following:

- Calls `flea_db_traverse` to move the vehicle if the user has activated the database traversal feature.
- Calls `flea_agl_terrain_follow` to adjust the vehicle's altitude if the user has activated the AGL terrain follow feature.
- Increments `flea_vppos.x`, `flea_vppos.y`, and `flea_vppos.z` by `skidx`, `skidy`, and `skidz`, respectively.
- Calls `flea_simulate_vehicles` to update the position and orientation of each Flea vehicle based on the terrain feedback points received.
- Increments `flea_vppos.x`, `flea_vppos.y`, and `flea_vppos.z` by speed times the applicable value in the `flea_htw_transform.mtx` array.
- If demonstration is not 0 (indicating that a demo is in progress), calls `flea_demo` to run the next frame of the autopilot demo.
- Checks all rotation values (`flea_vprot.x`, `y`, and `z`). If any is less than 0, adds 360. If any is greater than 360, subtracts 360.
- Uses the `CHECKROT` macro (described in Appendix B) to verify that all rotation values are now valid.
- Calculates the sine and cosine of each rotation value and places the values in `p_flea_out`.
- Calls `make4x3` to generate the simulated vehicle's new transformation matrix.
- Updates the values in the `flea_htw_transform.mtx` and `flea_htw_transform.vec` arrays.

Called By: `flea`

Routines Called: `CHECKROT`
`cos`
`flea_agl_terrain_follow`
`flea_db_traverse`
`flea_demo`
`flea_simulate_vehicles`
`make4x3`
`printf`
`sin`
`TORAD`

Parameters: `none`

Returns: none

2.7.23 flea_veh_control.c

The functions in `flea_veh_control.c` let the user move the vehicles in the simulation. This CSU contains the following functions:

- `flea_veh_control`
- `flea_veh_control_main_menu`

2.7.23.1 flea_veh_control

The `flea_veh_control` function processes commands entered by the user to move vehicles (the simulated vehicle or other dynamic vehicles in the exercise) through the terrain. This function is called via the `*flea_menu` function pointer when the user selects the ^ ("vehicle control") option from the Flea main menu.

`flea_veh_control` maintains an array (`p_flea_veh[]` — data type `SIM_VEH_STRUCT`) for the vehicles being manipulated. Each vehicle's entry specifies its type, vehicle id, ASID value, transformation matrix, desired altitude, terrain feedback points, etc. The first vehicle in the array (index 0) is the simulated vehicle. The maximum number of vehicles that can be controlled is specified by `MAX_FLEA_VEHICLES`, which is set in the `demo_struct.h` file.

The function call is `flea_veh_control()`. The function uses `*flea_getc` to get the character entered by the user, then processes the command.

The following table lists the options supported by `flea_veh_control`, and shows the major steps it performs to process each one. These options are displayed by `flea_veh_control_main_menu`. Options flagged with an asterisk are supported by `flea_veh_control` but do not appear on the menu.

Vehicle Control Menu Option	Processing by flea_veh_control
?* display menu	Calls flea_veh_control_main_menu.
! fire a round	If vehicle has no round type defined: prompts user for round and tracer types; sets vehicle's round_type and tracer_type. Sets vehicle's shoot_flag to TRUE.
, 0 z skid	Sets vehicle's dpos.z to 0.0.
. + z skid	Increments vehicle's dpos.z by 0.1.
: new orientation	(not currently implemented)
@ toggle autofire	If vehicle's round_limit is non-zero (autofire is on), sets round_limit to 0. If vehicle's round_limit is 0 (autofire is off): If vehicle has no round type specified, prompts user for round and tracer types; sets vehicle's round_type and tracer_type. Prompts user for rounds per minute; sets vehicle's round_limit = $60 * \text{frame_rate} / \text{rounds_per_minute}$.
1 + z rot	Increments vehicle's drot.z by 0.1.
2 0 z rot	Sets vehicle's drot.z to 0.0.
3 - z rot	Decrements vehicle's drot.z by 0.1.
A new veh asid	Prompts user for new vehicle ASID value; sets current vehicle's asid.
c change veh type	Prompts user for new vehicle type (in decimal); sets vehicle's vehicle_type.
d delete veh	Sets vehicle's clear_flag to TRUE.
h stop	Sets vehicle's dpos.y to 0.0.
i init veh	Clears screen. If vehicle's in_use_flag is TRUE, displays error message and exits. Prompts user for vehicle type, asid, x and y positions, desired altitude; sets values in vehicle's entry in p_flea_veh[]. If vehicle is the simulated vehicle: sets vehicle's z position to flea_vprot.z; sets vehicle's dpos.y to speed. Prompts user for x,y locations of three terrain feedback points; puts data in tf_init_pts[]; sets vehicle_id and tf_init_hdr.vehicle_id to index number from p_flea_veh[]; sets all terrain feedback parameters; sets vehicle's in_use_flag to TRUE.
M new veh type	Prompts user for new vehicle type (in hex); sets vehicle's vehicle_type.
m - z skid	Decrements vehicle's dpos.z by 0.1.
n go backward	Decrements vehicle's dpos.y by 0.1.
p new position	(not currently implemented)
q* quit	Sets *flea_menu function pointer to tick; sets current prompt to "Flea>"; calls tick_main_menu.

v change veh	Prompts user for vehicle number; verifies that number is less than MAX_FLEA_VEHICLES - if not, sets index to 0; sets pointer to vehicle; displays vehicle's address in p_flea_veh[] array.
x* exit	Sets *flea_menu function pointer to tick; sets current prompt to "Flea>"; calls tick_main_menu.
y go forward	Increments vehicle's dpos.y by 0.1.

Called By: flea_io_task (through *flea_menu)

Routines Called: *flea_getc
blank
cup
flea_veh_control_main_menu
GLOB
isprint
isspace
printf
scanf
strcpy
tick_main_menu

Parameters: none

Returns: none

2.7.23.2 flea_veh_control_main_menu

The flea_veh_control_main_menu function displays the Flea Vehicles menu processed by flea_veh_control. This function is called if the user selects the ^ ("vehicle control") option from the Flea main menu, or enters ? on the Flea Vehicles menu.

The function call is flea_veh_control_main_menu(). The function does the following:

- Clears the screen.
- Displays the following information on the simulated vehicle:
 - Vehicle id.
 - Vehicle type.
 - Current speed.
 - Current x, y, and z coordinates.
 - Current x, y, and z rotation values.
 - Terrain feedback "own vehicle" flag (TRUE or FALSE).
 - Desired altitude above ground.
 - Coordinates of the first three terrain feedback points.
- Displays the menu of available options.
- Copies "Flea Vehicles>" to the current prompt.
- Displays the current prompt.

For a list of the options displayed by `flea_veh_control_main_menu`, see `flea_veh_control`.

Called By: `flea_veh_control`
`tick`

Routines Called: `blank`
`cup`
`printf`
`strcpy`

Parameters: `none`

Returns: `none`

2.7.24 `get_sio_data.c` (`get_sio_write_data`)

The `get_sio_write_data` function lets the Flea user send data to a serial input/output device attached to the system. This function is called if the user selects the Z ("Write to Video Mux") option from the Flea AGPT Switches menu. `get_sio_write_data` prompts the user for the required parameters, stores the data, then sets a flag for `upd_sio_write`. `upd_sio_write` then generates the applicable `MSG_SIO_WRITE` message.

The function call is `get_sio_data()`. The function does the following:

- Prompts the user for the serial device id.
- Prompts the user for the string to be written to the device.
- Prompts the user for the frame delay (the number of frames to wait before sending the data).
- Puts the user's entries in `sio_write_message`.
- Sets `sio_flag` to `TRUE`.

Called By: `flea_agpt_switches`

Routines Called: `gets`
`printf`
`sscanf`
`strcat`
`strlen`

Parameters: `none`

Returns: `none`

2.7.25 `model_demo.c`

The `model_demo` function provides options to the user to control the moving model demonstration. The user can specify the model id to use, the number of models to display, and how far apart the models should be. The user can also stop the demo.

This function is called if the user selects the `d` ("moving model demo") option from the Flea Switches menu, or the `U` ("Moving Model Menu") option from the Flea AGPT Switches menu. It is called only if a model demo has already been initiated. If no demo is yet in progress, selecting these options starts a demo.

The function call is `model_demo()`. The function does the following:

- Clears the screen.
- Prompts the user to enter a command.
- Processes the user's request (see table below).

The model demonstration variables that can be configured by the user are maintained in `p_flea_out->demo_ptr`. `demo_ptr` is used by `dynamic_demo`, which actually runs the demonstration. The maximum number of models that can appear in one demo is controlled by `MAX_DEMO_MODELS`, which is defined in the `demo_struct.h` file.

The options listed on the Moving Models Demonstration menu are listed below. The second column shows the steps `model_demo` performs to process each selection.

Moving Models Demonstration Menu Option	Processing by model_demo
? view options	Displays menu.
1 number of models	Prompts user for number of models to display; sets num_models; if number exceeds MAX_DEMO_MODELS, asks for another value and sets again.
2 model type	Prompts user for model type to display; sets model_id.
3 models in x direction	Prompts user for number of models to display in x direction; sets num_m_x.
4 models in y direction	Prompts user for number of models to display in y direction; sets num_m_y.
5 models in z direction	Prompts user for number of models to display in z direction; sets num_m_z.
6 space between models in x direction	Prompts user for space (in meters) between models in x direction; sets spacing_x.
7 space between models in y direction	Prompts user for space (in meters) between models in y direction; sets spacing_y.
8 space between models in z direction	Prompts user for space (in meters) between models in z direction; sets spacing_z.
9 stop demonstration	Sets demonstration to 0; calls dynamic_demo with frame number set to 9999.
q exit	Exits.
x exit	Exits.

Called By: flea_agpt_switches
flea_switches

Routines Called: blank
cup
dynamic_demo
printf
scanf

Parameters: none

Returns: none

2.7.26 tick.c

The functions in the tick.c CSU handle the Flea main menu and general operations related to the Flea user interface. These functions are:

- tick_init
- tick
- flea_prompt
- tick_main_menu
- menu_header
- update_menu_header

2.7.26.1 tick_init

The tick_init function initializes the matrix type and transformation matrix for the simulated vehicle. This function is called when a Flea exercise is initiated, when Flea configuration is complete.

The function call is tick_init(). tick_init does the following:

- Calls flea_prompt to display the "Flea>" prompt.
- Initializes the information in the first dynamic node in p_flea_out using the viewpoint position and orientation entered by the user.
- Sets the p_flea_out->xfrm_update flag for index 0 to TRUE. This flag alerts upd_dynamic_matrix that a MSG_HPRXYZS_MATRIX message is required.
- Sets the p_flea_out->mtx_type for index 0 to RTS4x3_TYPE.

Called By: flea

Routines Called: flea_prompt

Parameters: none

Returns: none

2.7.26.2 tick

The tick function processes the options listed on the Flea main menu. This menu lets the user do the following:

- Access the other Flea menus.
- Move the simulated vehicle over the terrain.
- Invoke or stop database traversal.
- Control automatic updating of the data displayed at the top of the console screen.
- Return to Gossip.
- Stop the Flea exercise.

This function is called (via the *flea_menu function pointer) when the user starts Flea and after the user exits out of a lower-level Flea menu.

The function call is tick(). tick does the following:

- Uses the *flea_getc function pointer to get the keystroke entered by the user.

- Sets the heading of the simulation vehicle.
- Processes the user's keystroke (see table below).
- Displays the "Flea>" prompt.
- If the system is set for constant menu updating, Flea I/O mode is enabled, and the update_ticks interval has passed since the last update, calls update_menu_header to display updated statistics for the simulated vehicle. (Constant updating and the update_ticks interval are set using the "toggle menu update" option on the Flea main menu.)

The following table lists the options supported by tick, and identifies the major steps it performs to process each one. The menu itself is displayed by tick_main_menu. Options flagged with an asterisk are supported by tick but do not appear on the menu.

Flea Main Menu Option	Processing by tick
?* display menu	Calls tick_main_menu.
^ vehicle control	Sets *flea_menu function pointer to flea_veh_control; calls flea_prompt to display "Flea Vehicles>" prompt; calls flea_veh_control_main_menu.
! fire a round	Sets p_flea_out->shoot_flag to TRUE.
# ballistics	Sets *flea_menu function pointer to flea_bal_opts; calls flea_prompt to display "Flea Ballistics>" prompt; calls flea_bal_opts_main_menu.
\$ DBCALIB locations	Sets *flea_menu function pointer to flea_atp; calls flea_prompt to display "Flea ATP>" prompt; calls flea_atp_main_menu.
& DBTEST locations	Sets *flea_menu function pointer to flea_graphics_test; calls flea_prompt to display "Flea Graphics>" prompt; calls flea_graphics_test_main_menu.
(AGPT locations	Sets *flea_menu function pointer to flea_agpt_locations; calls flea_prompt to display "Flea AGPT Locations>" prompt; calls flea_agpt_locations_main_menu.
) AGPT switches	Sets *flea_menu function pointer to flea_agpt_switches; calls flea_prompt to display "Flea AGPT Switches>" prompt; calls flea_agpt_switches_main_menu.
* switches	Sets *flea_menu function pointer to flea_switches; calls flea_prompt to display "Flea Switches>" prompt; calls flea_switches_main_menu.
, 0z skid	Sets skidz to 0.
. +z skid	Increases skidz by .03.
: new orientation	Initializes htrate and speed to 0; prompts user for new viewpoint orientation; sets flea_vprot.x, y, and z. Validates rotation values: if any value is less than 0, adds 360; if any value is greater than 360, subtracts 360. Puts data in p_flea_out->orien[0]; sets vehicle's speed, skid, and rotation values in p_flea_out to 0.
1 +z rot	Increases htrate by 0.1.
2 0z rot	Sets htrate to 0.00.

3	-z rot	Decreases htrate by 0.1.
4	+x rot	Increases flea_x_rot by 0.1.
5	0x rot	Sets flea_x_rot to 0.0.
6	-x rot	Decreases flea_x_rot by 0.1.
7	+y rot	Increases flea_y_rot by 0.1.
8	0y rot	Sets flea_y_rot to 0.0.
9	-y rot	Decreases flea_y_rot by 0.1.
H	stop db trv	Sets p_flea_out->db_trv_state to 2 (halt traversal).
h	stop	Sets speed to 0.
i	0x skid	Sets skidx to 0.
j	-y skid	Decreases skidy by .03.
k	0y skid	Sets skidy to 0.
l	+y skid	Increases skidy by .03.
m	-z skid	Decreases skidz by .03.
n	reverse	Decreases speed by 0.2.
o	+x skid	Increases skidx by .03.
P	Calibrate PPMs	Sets *flea_menu function pointer to tick_ppm; calls flea_prompt to display "Flea PPM>" prompt; calls tick_ppm_menu.
p	new position	Initializes htrate and speed to 0; prompts user for new viewpoint position; sets flea_vppos.x, y, and z; prompts user for new viewpoint orientation; sets flea_vprot.x, y, and z; puts data in p_flea_out->xlate[0] and p_flea_out->orien[0]; sets vehicle's speed, skid, and rotation values in p_flea_out to 0.
q	quit (w/o stopping)	Calls flea_IO_off to disable Flea I/O; calls gos_IO_on to enable Gossip I/O; calls gos_prompt to display "Gossip>" prompt; calls flea_prompt to display "Gossip>" prompt.
R	resume db trv	Sets p_flea_out->db_trv_state to 4 (resume traversal).
r	set round type	Prompts user for type; sets p_flea_out->round_type.
S*	display db trv parameters	Clears screen; displays current database traversal state, direction, speed, interval, minimum x,y, and maximum x,y.

T start/restart db trv	<p>Clears screen; sets p_flea_out->db_trv_pos.x, y, and z to vehicle's current position; sets p_flea_out->db_trv_orient.x, y, and z to vehicle's current orientation.</p> <p>Prompts user for traverse direction; sets p_flea_out->db_trv_direction.</p> <p>Prompts user for traversal speed in meters per frame; sets p_flea_out->db_trv_speed.</p> <p>Prompts user for traversal interval in meters; sets p_flea_out->db_trv_interval.</p> <p>Prompts user for traversal minimum and maximum x and y coordinates; sets p_flea_out->db_trv_min_x, db_trv_max_x, db_trv_min_y, and db_trv_max_y.</p> <p>Sets p_flea_out->db_trv_state to 3 (turn traversal on).</p>
t toggle menu update	<p>If constant_update is FALSE (updating is disabled): prompts user for number of ticks between updates; sets update_ticks; sets constant_update to TRUE.</p> <p>If constant_update is TRUE (updating is enabled): sets constant_update to FALSE.</p>
u -x skid	Decreases skidx by .03.
x exit (stopping)	Sets skidx, skidy, skidz, htrate, and speed to 0; sets flea_x_rot and flea_y_rot to 0.0; sets all indices in p_flea_out->xfrm_update[] to FALSE.
y forward	Increases speed by 0.2.
Z scripting	Sets *flea_menu function pointer to tick_script; calls flea_prompt to display "Flea Script>" prompt; calls tick_script_main_menu.
z stop Flea (C_STOP)	Sets constant_update to FALSE; sets p_flea_out->stop to TRUE; calls flea_IO_off to display Flea I/O; calls gos_IO_on to enable Gossip I/O; calls gos_prompt to display "Gossip>" prompt; calls flea_prompt to display "Gossip>" prompt.

Called By: flea_io_task (through *flea_menu)

Routines Called:

- *flea_getc
- blank
- cup
- flea_agpt_locations_main_menu
- flea_agpt_switches_main_menu
- flea_atp_main_menu
- flea_bal_opts_main_menu
- flea_graphics_test_main_menu
- flea_IO_mode
- flea_IO_off
- flea_prompt
- flea_switches_main_menu
- flea_vch_control_main_menu
- GLOB
- gos_IO_on

gos_prompt
isprint
isspace
printf
scanf
tick_main_menu
tick_ppm_menu
tick_script_main_menu
update_menu_header

Parameters: none

Returns: none

2.7.26.3 flea_prompt

The `flea_prompt` function displays the Flea command prompt at the bottom of the user's screen. The prompt contains a text string that identifies the Flea menu that is currently active (e.g., "Flea>" for the main menu and "Flea Ballistics>" for the `flea_bal_opts` menu). The user enters the desired command next to the prompt.

The function call is `flea_prompt (newprompt)`, where *newprompt* is the character string to be displayed.

Called By: gossip_tick
tick
tick_init
tick_main_menu

Routines Called: cup
printf

Parameters: char *newprompt

Returns: none

2.7.26.4 tick_main_menu

The `tick_main_menu` function displays the Flea main menu. The options on this menu are used to move the simulated vehicle over the terrain, and to access the lower-level Flea menus. This function is called when the user first starts Flea, enters ? at the "Flea>" prompt, or exits out of a lower-level Flea menu.

The function call is `tick_main_menu()`. The function does the following:

- Clears the screen.

- Calls menu_header to display the labels for the Flea simulation statistics shown at the top of the screen.
- Displays the main menu.
- Calls flea_prompt to display the "Flea>" prompt.
- Calls update_menu_header to update the statistics at the top of the screen.

The option selected by the user is processed by the tick function. Refer to tick for a list of the options displayed by tick_main_menu.

Called By: flea_agpt_locations
 flea_agpt_switches
 flea_atp
 flea_bal_opts
 flea_graphics_test
 flea_switches
 flea_veh_control
 tick
 tick_ppm
 tick_script

Routines Called: blank
 cup
 flea_prompt
 menu_header
 printf
 update_menu_header

Parameters: none

Returns: none

2.7.26.5 menu_header

The menu_header function displays the labels for the current position, speed, and orientation of the simulation vehicle. The value for each parameter is displayed by the update_menu_header function. Lines 1 and 2 of the Flea console screen are used for this purpose.

The function call is menu_header().

Called By: flea_agpt_locations_main_menu
 flea_agpt_switches_main_menu
 flea_bal_opts_main_menu
 flea_switches_main_menu
 tick_main_menu
 tick_script_main_menu

Routines Called: cup
 printf

Parameters: none

Returns: none

2.7.26.6 **update_menu_header**

The `update_menu_header` function displays the current position, speed, and orientation (heading, pitch, and roll) of the simulation vehicle. The values are displayed on lines 1 and 2 of the Flea console, next to the labels displayed by `menu_header`. `update_menu_header` is called to update the values during the Flea exercise if constant updating is enabled. (Constant updating and the interval between updates are set using the "toggle menu update" option on the Flea main menu.)

The function call is **`update_menu_header()`**. The function does the following:

- Converts the simulation vehicle's current speed to knots.
- Displays the current values of `flea_vppos.x`, `flea_vppos.y`, `flea_vppos.z` next to the "Position" label.
- Displays the current speed next to the "Knots" label.
- Adjusts the vehicle's orientation values if outside the valid range (0-360).
- Displays the current/adjusted values of `flea_vprot.z`, `flea_vprot.x`, and `flea_vprot.y` next to the "Heading," "Pitch," and "Roll" labels.

Called By: **`flea_agpt_locations_main_menu`**
 `flea_agpt_switches_main_menu`
 `flea_bal_opts_main_menu`
 `flea_switches_main_menu`
 `tick`
 `tick_main_menu`

Routines Called: cup
 printf

Parameters: none

Returns: none

2.7.27 **tick_ppm.c**

The functions in the `tick_ppm.c` CSU are used to calibrate the PPM (Pixel Processor Memory) boards in the CIG. These functions are:

- `tick_ppm`

- tick_ppm_menu
- tick_ppm_menu_header
- tick_ppm_update_info

This CSU also defines and uses the SET_PPM_DISPLAY_OFFSET and SET_PPM_PIXEL_LOCATION macros, described in Appendix B.

2.7.27.1 tick_ppm

The tick_ppm function processes the commands entered by the user on the Flea PPM menu. This function is called via the *flea_menu function pointer if the user selects the P ("Calibrate PPMs") option from Flea main menu. tick_ppm obtains the user's entries, puts the data in p_flea_out, and sets various flags. The messages required to make the changes are generated at the end of the frame by functions in the flea_ppm_obj.c CSU.

The function call is tick_ppm(). The function does the following:

- Uses the *flea_getc function pointer to get the keystroke entered by the user.
- Processes the user's request (see table below).
- Displays the "Flea PPM>" prompt.
- Every 150 frames, calls tick_ppm_menu to display the Flea PPM menu.
- Every 15 frames (unless also a multiple of 150), calls tick_ppm_update_info to refresh the display of PPM parameter data.

The following table lists the commands supported by tick_ppm, and identifies the steps it uses to process each one. The menu that lists these commands is displayed by tick_ppm_main_menu. Options flagged with an asterisk (*) are supported by tick_ppm but do not appear on the menu.

Flea PPM Menu Option	Processing by tick_ppm
?* display this menu	Calls tick_ppm_menu.
1 display left	Decrements i value in display_offsets array for current backend/channel; calls SET_PPM_DISPLAY_OFFSET macro to put data in p_flea_out.
2 display down	Increments j value in display_offsets array for current backend/channel; calls SET_PPM_DISPLAY_OFFSET macro to put data in p_flea_out.
3 display right	Increments i value in display_offsets array for current backend/channel; calls SET_PPM_DISPLAY_OFFSET macro to put data in p_flea_out.
5 display up	Decrements j value in display_offsets array for current backend/channel; calls SET_PPM_DISPLAY_OFFSET macro to put data in p_flea_out.
A* stop auto test	Sets p_flea_out->auto_ppm_flag to FALSE.

a auto test	Clears screen; sets p_flea_out->auto_ppm_backend to current backend id; prompts user for display mode, display offset, calibration pixel, and calibration pixel location periods; sets p_flea_out->display_mode_period, display_offset_period, cal_pixel_period, and pixel_loc_period; sets p_flea_out->auto_ppm_flag to TRUE; calls tick_ppm_menu.
b change backend	Prompts user for backend id; sets backend; calls tick_ppm_menu.
c change channel	Prompts user for channel number; sets channel; calls tick_ppm_menu.
h pixel left	Outputs error if channel >= 4; decrements i value in pixel_location array for current backend/channel; calls SET_PPM_PIXEL_LOCATION macro to put data in p_flea_out.
i cal image	Sets p_flea_out->calibration_image.subsystem; prompts user for image number; sets p_flea_out->calibration_image.image; sets p_flea_out->calibration_image_flag to TRUE; calls tick_ppm_menu.
j pixel down	Outputs error if channel >= 4; increments j value in pixel_location array for current backend/channel; calls SET_PPM_PIXEL_LOCATION macro to put data in p_flea_out.
k pixel up	Outputs error if channel >= 4; decrements j value in pixel_location array for current backend/channel; calls SET_PPM_PIXEL_LOCATION macro to put data in p_flea_out.
l pixel right	Outputs error if channel >= 4; increments i value in pixel_location array for current backend/channel; calls SET_PPM_PIXEL_LOCATION macro to put data in p_flea_out.
m change mode	Prompts user for new mode; sets p_flea_out->ppm_display_mode.subsystem, channel, and display_mode; displays values; sets p_flea_out->ppm_display_mode_flag to TRUE; calls tick_ppm_menu.
O pixel on	Outputs error if channel >= 4; sets p_flea_out->ppm_pixel_state.subsystem and channel; sets p_flea_out->ppm_pixel_state.state to TRUE; sets p_flea_out->ppm_pixel_state_flag to TRUE.
o pixel off	Outputs error if channel >= 4; sets p_flea_out->ppm_pixel_state.subsystem and channel; sets p_flea_out->ppm_pixel_state.state to FALSE; sets p_flea_out->ppm_pixel_state_flag to TRUE.
q* quit	Sets *flea_menu function pointer to tick; calls tick_main_menu.
x* exit	Sets *flea_menu function pointer to tick; calls tick_main_menu.

Called By: flea_io_task (through *flea_menu)

Routines Called: *flea_getc
 blank
 cup
 isprint
 isspace
 printf
 scanf
 SET_PPM_DISPLAY_OFFSET
 SET_PPM_PIXEL_LOCATION
 tick_main_menu
 tick_ppm_menu
 tick_ppm_update_info

Parameters: none

Returns: none

2.7.27.2 **tick_ppm_menu**

The `tick_ppm_menu` function displays the Flea PPM menu. This function is called by `tick` if the user selects the **P** ("Calibrate PPMs") option from the Flea main menu. It is called by `tick_ppm` after certain options are processed, if the user enters `?` at the "Flea PPM>" prompt, and every 150 frames.

The function call is `tick_ppm_menu()`. The function does the following:

- Clears the Flea console screen.
- Calls `tick_ppm_menu_header` to display the labels for the PPM parameters.
- Calls `tick_ppm_update_info` to display the current value of each PPM parameter.
- Displays the Flea PPM menu.
- Displays the "Flea PPM>" prompt.

The options listed on this menu are processed by the `tick_ppm` function. For a list and description of the options, see `tick_ppm`.

Called By: tick
 tick_ppm

Routines Called: blank
 cup
 printf
 tick_ppm_menu_header
 tick_ppm_update_info

Parameters: none

Returns: none

2.7.27.3 tick_ppm_menu_header

The `tick_ppm_menu_header` function displays the labels for the current PPM calibration settings shown at the top of the Flea console screen when the Flea PPM menu is active. The current value for each parameter (backend id, channel number, PPM mode, display offset, and pixel location) is displayed by `tick_ppm_update_info`.

`tick_ppm_menu_header` also displays the menu title ("FLEA PPM MENU").

The function call is `tick_ppm_menu_header()`.

Called By: `tick_ppm_menu`

Routines Called: `cup`
`printf`

Parameters: none

Returns: none

2.7.27.4 tick_ppm_update_info

The `tick_ppm_update_info` function displays the current value of each PPM calibration parameter (backend id, channel number, PPM mode, x and y display offsets, and x and y pixel locations) at the top of the Flea console screen when the Flea PPM menu is active. This function is called when the menu is first displayed. It is also called to update the data every 15 frames, as long as the menu is still active.

The label for each PPM calibration parameter is displayed by `tick_ppm_menu_header`.

The function call is `tick_ppm_update_info()`.

Called By: `tick_ppm`
`tick_ppm_menu`

Routines Called: `cup`
`printf`

Parameters: none

Returns: none

2.7.28 tick_script.c

The functions in the tick_script.c CSU provide options that let the Flea user run pre-built scripts to drive the simulation. These scripts, called playback scripts, are created using the r ("record i/f messages menu") option on the Gossip main menu. If a script is selected for playback, it is processed by flea_abs_playback.

The functions in this CSU are:

- tick_script
- tick_script_menu

2.7.28.1 tick_script

The tick_script function processes the options displayed on the Flea Script menu. These options let the Flea user run playback scripts (recorded by the cigsimio functions in the Real-Time Processing CSC). tick_script is called via the *flea_menu function pointer if the user selects the Z ("scripting") option from the Flea main menu.

The function call is tick_script(). The function does the following:

- Uses the *flea_getc function pointer to get the keystroke entered by the user.
- Processes the user's request (see table below).
- Displays the current prompt ("Flea Script>").

The following table lists the options supported by tick_script, and shows the steps it uses to process each one. The menu that lists these options is displayed by tick_script_main_menu. Options flagged with an asterisk (*) are supported by tick_script but do not appear on the menu.

Flea Script Menu Option	Processing by tick_script
? menu	Calls tick_script_main_menu.
B blank screen	Calls blank to clear screen.
b begin absolute playback	If playback file has been set (i.e, playback_file_set is TRUE), sets fleaG_abs_playback to TRUE. If playback file has not been set, outputs error.
e stop playback	Sets fleaG_stop_playback to TRUE.
f specify playback file	Prompts user for name of playback file; sets fleaG_playback_fn; opens the file; sets playback_file_set to TRUE.
q* quit	Sets *flea_menu function pointer to tick; sets current prompt to "Flea>"; calls tick_main_menu.
x exit	Sets *flea_menu function pointer to tick; sets current prompt to "Flea>"; calls tick_main_menu.

Called By: flea_io_task (through *flea_menu)

Routines Called: *flea_getc
 blank
 cup
 ifx_close
 ifx_open
 isprint
 isspace
 printf
 scanf
 strcpy
 tick_main_menu
 tick_script_main_menu

Parameters: none

Returns: none

2.7.28.2 tick_script_main_menu

The tick_script_main_menu function displays the menu supported by tick_script. This function is called by tick if the user selects the Z ("scripting") option from the Flea main menu. It is called by tick_script if the user enters ? at the "Flea Script>" prompt.

The function call is **tick_script_main_menu()**. The function does the following:

- Clears the Flea console screen.
- Calls menu_header to display the standard Flea status fields.
- Displays the Flea Script options.
- Sets the current prompt to "Flea Script>".
- Displays the "Flea Script>" prompt.

Refer to tick_script for a list and descriptions of the options displayed on the Flea Script menu.

Called By: tick
 tick_script

Routines Called: blank
 cup
 menu_header
 printf
 strcpy

Parameters: none

Returns: none

2.7.29 update_2d.c

The `update_2d` function generates a `MSG_PASS_ON` message to send a request to the 2-D processor task in a specified subsystem. This function is called every frame by `flea_encode_data` if the (1) subsystem's element in the `sub_2d_io[]` array is set to `TRUE`, and (2) `type_update_2d` is not 1.

The subsystem's flag in `sub_2d_io[]` is set by `flea_switches` if the user selects the 2 ("2d mode") option from the Flea Switches menu. `flea_switches` also places the command selected by the user into the subsystem's element in the `command_2d[]` array, and sets the subsystem's flag in the `gsp_io_flag[]` array to `TRUE`. The entry in `gsp_io_flag[]` tells the `mpvideo_send_req` function (in the MPV Interface CSC) that there is a command to be sent to the GSP in this subsystem.

The `type_update_2d` flag is set to 1 only if the user selects a 2-D option from the Flea AGPT Switches menu. If the variable is not 1, `flea_encode_data` assumes the option was selected from the regular Flea Switches menu, and calls `update_2d`.

The function call is `update_2d(subsystem)`, where *subsystem* is the backend id. The function does the following:

- Determines the command in `command_2d[subsystem]`.
- Builds a message header for the `MSG_PASS_ON` message, with the message length determined by the command.
- Builds the `MSG_PASS_ON` message, based on the command.

Called By: `flea_encode_data`

Routines Called: `GLOB`

Parameters: `INT_4` subsystem

Returns: none

2.7.30 update_agpt_2d.c

The `update_agpt_2d` function generates a `MSG_PASS_ON` message to send a request to the 2-D processor task in a specified subsystem. This function is called every frame by `flea_encode_data` if (1) the subsystem's element in the `sub_2d_io[]` array is set to `TRUE`, and (2) the `type_update_2d` flag is 1.

The subsystem's flag in `sub_2d_io[]` is set by `set_command_2d` if the user selects "Aliasing Model (2D)," "2D Dimmer," "2D Brighter," or any of the "Reticle" options from the Flea AGPT Switches menu. `set_command_2d` also places the command to be executed into the subsystem's element in the `command_2d[]` array, and sets `type_update_2d` to 1.

The function call is **update_agpt_2d(subsystem)**, where *subsystem* is the backend id. The function does the following:

- Determines the command in the command_2d[] array.
- Builds a message header for the MSG_PASS_ON message, with the message length determined by the command.
- Builds the MSG_PASS_ON message, based on the command.

Called By: flea_encode_data

Routines Called: none

Parameters: INT_4 subsystem

Returns: none

2.8 DTP Command Generator (/cig/libsrc/libgendtp)

The DTP (Data Traversal Processor) Command Generator translates the viewport configuration tree generated by the Viewport Configuration CSC into the commands required to drive the graphics hardware. It generates DTP hardware commands (processor and channel initialization code) from the viewport configuration tree, then downloads these commands to the DTP CPU. The DTP then determines what data is to be sent to the 9U graphics channel.

The DTP is a micro-coded processor board that does the following:

- Looks through active area memory for DTP commands.
- Computes viewpoint positions for vectors.
- Computes world-to-viewpoint matrices for each viewport.
- Performs field-of-view and level-of-detail tests on models and special effects.
- Sends data to the Polygon Processor.

The Polygon (Poly) Processor is a special-purpose floating point processor that does the following:

- Transforms polygons from world coordinates to viewspace coordinates.
- Eliminates back-facing polygons.
- Clips polygons that fall partially outside of the viewing pyramid.
- Fills polygons with colored or textured pixels.
- Perspectively projects polygons onto the screen.

The DTP is controlled through the DTP commands it finds in active area memory. These commands are placed in active area memory by the DTP Command Generator. The DTP reads one buffer in double-buffer memory while the real-time software updates the other. Each frame, the two processes switch buffers.

The DTP Command Generator uses the Runtime Command Library (RCL) to generate DTP commands. The RCL is a set of software functions that support the configuration of lists of runtime commands for both the DTP and the Poly Processor. The RCL is responsible for working with the complex data structures in the DTP — the DTP Command Generator simply specifies the command and provides the data required for the command. The RCL also maintains addressing and data sizing information.

The interface between the DTP Command Generator functions and the RCL is implemented via command-specific macros. Each DTP command is supported by one or more macros. These macros are named in the form *dtp_xyz*, where *xyz* identifies the DTP command or a version of a command. Similarly, macros that support Poly Processor commands are named in the form *poly_xyz*. The DTP Command Generator function calls the appropriate macro and passes it the data required for the selected command. The macro in turn calls the appropriate RCL routine and passes it the command parameters. The RCL routine then generates the actual DTP command and places it in active area memory.

The DTP-RCL macros are defined in the *rcinclude.h* file. Refer to Appendix B for a list of these macros.

Figure 2-12 identifies the CSUs in the DTP Command Generator CSC. These CSUs are described in this section.

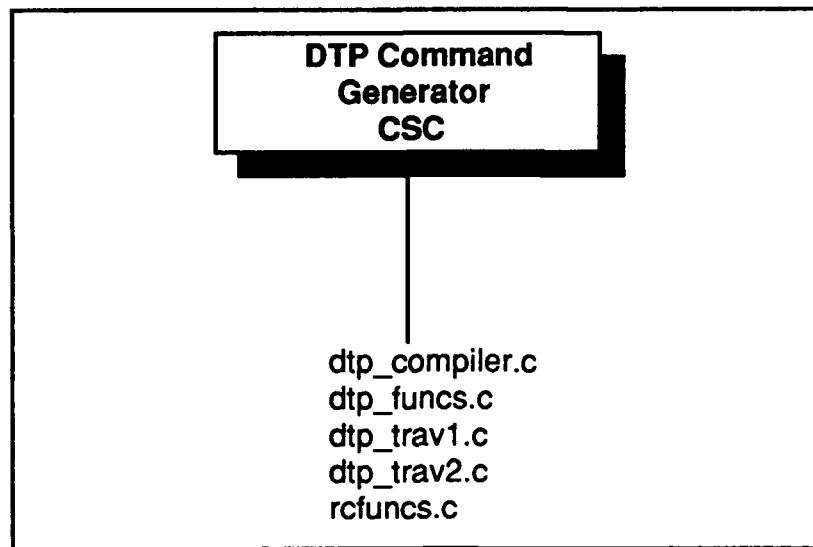


Figure 2-12. DTP Command Generator CSUs

2.8.1 `ntp_compiler.c`

The `ntp_compiler` function is the driving function for generating DTP hardware commands from the viewport configuration tree.

The function call is `ntp_compiler(offset)`, where *offset* is the number of bytes of DTP code. `ntp_compiler` does the following:

- Initializes the runtime command library (RCL), as follows:
 - Calls `malloc` to allocate memory for the RCL stack.
 - Calls `rcl_set_errptr` to set the error pointer to "`ntp_compiler`."
 - Calls `rcl_init_adrs` to initialize shared addressing variables.
 - Calls `rcl_init_stack` to initialize the RCL stack.
- Allocates data pointers for model processing.
- Starts the processor initialization phase, as follows:
 - Calls `rcl_set_errptr` to set the error pointer to "`ntp_trav1`."
 - Calls `init_ntp_stacks` to initialize the DTP stack.
 - Calls `vpt_cnode_qroot` to get a pointer to the configuration tree's root node for `ntp_trav1`.
 - Calls `ntp_trav1` to traverse the configuration tree for processor initialization.
 - Calls `rcl_patch_adrs` to run the RCL patch utility to patch any missing addresses and word counts.
 - Calls `rcl_init_stack` to reinitialize the RCL stack.
- Starts the channel initialization phase, as follows:
 - Calls `rcl_set_errptr` to set the error pointer to "`ntp_trav2`."
 - Calls `init_ntp_stacks` to initialize the DTP stack.
 - Calls `ntp_trav2` to traverse the configuration tree for channel initialization.
 - Calls `rcl_patch_adrs` to run the RCL patch utility again.
- Calls `rcl_rtn_adrs` to get the current RCL addressing values.
- Prints DTP memory use data.

- Frees the RCL stack.

The function returns SUCCEED if the commands are generated successfully. It returns FAIL if stack memory could not be allocated, the root node could not be found, or either dtp_trav1 or dtp_trav2 failed.

Called By: cig_config

Routines Called: dtp_trav1
 dtp_trav2
 free
 init_dtp_stacks
 malloc
 printf
 rcl_init_adrs
 rcl_init_stack
 rcl_patch_adrs
 rcl_rtn_adrs
 rcl_set_errptr
 vpt_cnode_qroot

Parameters: WORD offset

Returns: 1 (SUCCEED)
 0 (FAIL)

2.8.2 dtp_funcs.c

The functions in the dtp_funcs.c CSU are used to manage the node stack used when traversing the configuration tree, and to allocate DTP user memory. These functions are:

- push_node
- pop_node
- what_node_on_stack
- init_dtp_stacks
- dtp_malloc
- dtp_malloc_init

2.8.2.1 push_node

The push_node function takes a configuration node pointer as input and places it on the node stack. It also checks for and reports stack overflows.

The function call is **push_node(node_ptr)**, where *node_ptr* is a pointer to the configuration node to be pushed on the top of the stack.

Called By: dtp_trav1

Routines Called: printf

Parameters: CFG_NODE *node_ptr

Returns: none

2.8.2.2 pop_node

The `pop_node` function returns the configuration node pointer from the top of the node stack. If the stack is empty, `pop_node` returns 0; this tells `ntp_trav1` that the stack has been processed completely.

The function call is `pop_node()`.

Called By: ntp_trav1

Routines Called: none

Parameters: none

Returns: node_stack[--top_node_stack]
 0

2.8.2.3 what_node_on_stack

The `what_node_on_stack` function returns the index of the node on top of the node stack. If the stack is empty, the function returns a value specified by the calling procedure.

The function call is `what_node_on_stack(empty)`, where *empty* is the value to be returned if the stack is empty.

Called By: ntp_trav1

Routines Called: none

Parameters: UNS_4 empty

Returns: node_stack[top_node_stack - 1]->node_index
 empty

2.8.2.4 init_dtp_stacks

The `init_dtp_stacks` function initializes the node stack by setting the top stack pointer to 0. This function is called by `dtp_compiler` before it calls `dtp_trav1` to traverse the configuration tree.

The function call is `init_dtp_stacks()`.

Called By: `dtp_compiler`

Routines Called: none

Parameters: none

Returns: none

2.8.2.5 dtp_malloc

The `dtp_malloc` function allocates a specified amount of DTP memory. This function is called by `dtp_trav1` to allocate memory for configuration node matrices.

The function call is `dtp_malloc(count)`, where *count* is the amount of memory to be allocated.

The function returns 0 if successful. It returns the current DTP user pointer (as *give_away*) if insufficient memory is available.

Called By: `dtp_trav1`

Routines Called: none

Parameters: `INT_2` `count`

Returns: 0
give_away

2.8.2.6 dtp_malloc_init

The `dtp_malloc_init` function initializes the portion of DTP allocated as user space. It sets the DTP user pointer to the first available memory location, which is defined in `ecompile1.h`. `dtp_trav1` calls this function before it starts traversing the configuration tree.

The function call is **ntp_malloc_init()**.

Called By: **ntp_trav1**

Routines Called: **none**

Parameters: **none**

Returns: **none**

2.8.3 **ntp_trav1.c**

The **ntp_trav1** function traverses the configuration tree to generate processor initialization codes. **ntp_trav1** traverses each node in the configuration tree by placing the root node on the stack and then processing the stack until it is empty. When a node is popped from the stack, any matrix concatenation commands or bit tests are performed for that node, based on the node's type. The node's children and siblings are then placed on the stack such that the order of processing is the node, the node's children, and the node's siblings.

ntp_trav1 uses the routines in **ntp_funcs.c** to access and manage the node stack. It uses the **ntp_*** macros (defined in Appendix B) to communicate with the RCL to generate the actual commands for the hardware.

The function call is **ntp_trav1(node)**, where *node* is a pointer to the configuration tree's root node.

ntp_trav1 does the following:

- Calls **ntp_malloc_init** to initialize the DTP user space.
- Uses various **ntp_*** macros to load the following:
 - Channel status and channel pointers at DTP location 0.
 - List of final processing.
 - Flush and dynamic pointer tables.
 - Calibration branch mask.
 - Cloud bottom and top branch masks (if enabled).
 - Daylight TV thermal word.
 - View mode word for each channel.
 - System view flags and branch values.
 - Current time set in simulation.
- Processes each node in the tree to generate the matrix concatenations and bit tests for each path, as follows:
 - Calls **push_node** to push the root child 0 on the stack.
 - Calls **pop_node** to pop each node from the stack in turn.
 - Calls **rcl_set_label** to set a label for the node.
 - Validates the node's parent pointer.
 - For a matrix node:
 - * Allocates DTP memory for the node's matrix.
 - * Concatenates the matrix with the parent's matrix.

- For a branch/matrix node:
 - * Test the node's branch value.
 - * Allocate DTP memory for the node's matrix.
 - * If the branch value is true, load the node's matrix or concatenate it with the parent's matrix.
 - * If the branch value is false, load the parent's matrix.
- For a branch (conditional) node:
 - * Test the node's branch value.
 - * Load the parent's matrix.
- Push the node's siblings and children onto the stack.
- Performs initial data traversal.
- Prepares system post-processing pointers and displays the post-processing addresses for static vehicles, dynamic vehicles, and effects.
- Allocates space for the current time to support time-base commands.
- Calls rcl_data to generate a command to indicate a separation of initialization and channel processing.

The function returns 1 if successful. It returns 0 if it detects an illegal parent pointer or an invalid node type.

Called By: dtp_compiler

Routines Called:

- be_query_db0
- dtp_bnz
- dtp_bru
- dtp_brus
- dtp_end
- dtp_lwd
- dtp_lwds
- dtp_malloc
- dtp_malloc_init
- dtp_mmpst
- dtp_mwd
- poly_flu
- pop_node
- printf
- push_node
- rcl_data
- rcl_rtn_adrs
- rcl_set_errptr
- rcl_set_label
- what_node_on_stack

Parameters: CFG_NODE *node

Returns: 0
1

2.8.4 dtp_trav2.c

The `dtp_trav2` function traverses the configuration tree to generate channel initialization codes.

The function call is `dtp_trav2(node)`, where *node* is a pointer to the root configuration node.

`dtp_trav2` does the following:

- Saves the beginning path location.
- For a branch (conditional) node:
 - Tests the condition.
 - Traverses the true path.
- For a matrix node that is the terminal node in a traversal path (i.e., a node that has viewport parameters):
 - Calculates the channel base address.
 - Loads the channel parameters, field-of-view vectors, viewpoint position, level-of-detail multiplier, and far plane.
 - Multiplies the hull-to-view matrix for DTP use.
 - Calculates bounding plane normals.
 - Calculates the base load module.
 - Outputs the channel toggle command if the channel is secondary.
 - Outputs the perspective matrix.
 - Outputs the screen constants.
 - Tests for *calibration output* for all screens.
 - Outputs the gun overlay if bit 0 of the node's branch mask is set.
- For the root node:
 - Saves the matrix and forms the stamp word.
 - Calls the cloud top and bottom models, if enabled.
- Pre-processes models:
 - Creates `output_direct` for the node's matrix.
 - Outputs the gun barrel overlay if bit 1 of the node's branch mask is set.
 - For a branch node, sets the branch mask.
- Prepares the system pre-processing pointers and displays the pre-processing addresses for dynamic vehicles, static vehicles, and effects.
- Saves common poly command data.

The function always returns 1.

Called By: `dtp_compiler`

Routines Called: `be_qulm`
`dtp_blm`
`dtp_bnz`
`dtp_bpc`
`dtp_bru`
`dtp_brus`
`dtp_brz`
`dtp_end`

dtp_lwds
dtp_mmpst
dtp_osd
dtp_owd
dtp_owds
dtp_subs
poly_fsw
poly_rm1
poly_sm1
poly_tog
printf
rcl_rtn_adrs
rcl_set_errptr
rcl_set_label
rcl_stuff_data

Parameters: CFG_NODE *node

Returns: 1

2.8.5 rfuncs.c

The functions in the rfuncs.c CSU are used to work with the Runtime Command Library (RCL). These functions are:

- rcl_init_stack
- rcl_push
- rcl_pop
- rcl_patch_adrs
- rcl_set_errptr
- rcl_init_adrs
- rcl_rtn_adrs
- rcl_set_label
- rcl_set_cntlbl
- rcl_lblcmd
- rcl_command
- rcl_component
- rcl_data
- rcl_stuff_data

This CSU also defines the following macros used by the RCL functions. These macros are described in Appendix B.

- ERRMSG
- ROOM4LABEL
- ROOMCHECK
- INCR_COMPONENT

The RCL labeling utility removes the need for the programmer to maintain addressing and data size information as a command sequence is constructed. The programmer can write runtime code and label only data that is unknown. All labels (defined as single-integer

values) must uniquely identify one location in the code. As the library generates the runtime commands, it places any unknown information onto a patch stack. When the library encounters a label, it stores the location of the label for use in patching the stack. The `rcl_patch_adrs` function scans the list of unknown data and patches the missing addresses and word counts.

Use of the patching utility requires a stack area for maintaining the unresolved addresses, counts, and labels. The `rcl_init_stack` function is used to initialize this stack.

Most labels are used to identify a location in active area memory. Some labels are branch labels where DTP branch commands change the direction in which the DTP is processing messages. DTP output commands reference a location where the data begins. For these commands, the calling function specifies a unique label to identify the branch of output data, and uses the `rcl_set_label` function to identify the location. These locations are patched with the supplied addresses when the `rcl_patch_adrs` function is executed.

Set count labels are labels that are used to identify the size of a data segment rather than the location of command data.

The DTP has several output commands that require a word count value in order for the DTP to pass the correct amount of data to the Poly Processor. Usually, there are two ways to accomplish this:

- If the exact amount of data that can be sent is known, the DTP output command using the function that has data start label and word count parameters can be used.
- If the data size is not known, the command can be implemented using the set count function. Rather than specifying a word count for the command, a set count label is defined. When generating the data, `rcl_set_label` is executed to identify the beginning of the data. After generating the data, `rcl_set_cntlbl` is executed to specify the start and end labels, and the set count label is loaded with the word count of the data segment. When `rcl_patch_adrs` is executed, the output count is patched with the data segment size.

The DTP supports two addressing modes: absolute and relative. In absolute mode, the address is the actual AAM address for the branch or data. In relative mode, the address is an offset that is added to the current location to locate the branch or data.

2.8.5.1 `rcl_init_stack`

The `rcl_init_stack` function initializes the unresolved address, count, and label stack.

The function call is `rcl_init_stack(min_stack, max_stack)`, where:

min_stack is the minimum stack address (top of stack)

max_stack is the maximum stack address (bottom of stack)

Called By: `dtp_compiler`
 `gos_model`

Routines Called: none

Parameters:	UNS_4	*min_stack
	UNS_4	*max_stack

Returns:	none
----------	------

2.8.5.2 rcl_push

The `rcl_push` function adds a patch location to the patch stack, after verifying that the stack has space available.

The function call is `rcl_push(adr, lbladr, name)`, where:

adr is the physical memory address the patch is to be made in
lbladr is the physical memory address the label for the patch is in
name is the name of the calling routine

The function returns 0 if successful, or 1 if the stack is full.

Called By:	rcl_lblcmd
------------	------------

Routines Called:	ERRMSG
------------------	--------

Parameters:	UNS_4	*adr
	UNS_4	*lbladr
	char	*name

Returns:	0
	1

2.8.5.3 rcl_pop

The `rcl_pop` function removes the top patch location from the patch stack, after verifying that the stack is not empty.

The function call is `rcl_pop(adr, lbladr, name)`, where:

adr is a pointer to the physical memory address the patch is to be made in
lbladr is a pointer to the physical memory address the label for the patch is in
name is the name of the calling routine

The function returns 0 if successful, or 1 if the stack is empty.

Called By:	rcl_patch_adrs
------------	----------------

Routines Called: ERRMSG

Parameters: UNS_4 *adr
UNS_4 *lbladr
char *name

Returns: 0
1

2.8.5.4 rcl_patch_adrs

The `rcl_patch_adrs` function removes all entries from the patch stack, one at a time. It patches the stored location with the associated label location and processes the stack until it is empty. This function patches both absolute and relative addresses.

The function call is `rcl_patch_adrs()`.

Called By: dtp_compiler
gos_model
replace_mod
test_commands

Routines Called: ERRMSG
printf
rcl_pop

Parameters: none

Returns: none

2.8.5.5 rcl_set_errptr

The `rcl_set_errptr` function can be used to specify a character string to be output with every RCL error message. This string can help localize the source of the error.

The function call is `rcl_set_errptr(adr)`, where *adr* is the error string.

Called By: dtp_compiler
dtp_trav1
dtp_trav2
replace_mod

Routines Called: none

Parameters: char *adr

Returns: none

2.8.5.6 **rcl_init_adrs**

The **rcl_init_adrs** function initializes values for shared addressing variables.

The function call is **rcl_init_adrs(p1, p2, p3)**, where:

p1 is the memory location to store the RCL commands

p2 is the AAM location corresponding to the *p1* address

p3 is the number of bytes available for RCL commands, starting at the *p1* address

Called By: dtp_compiler
 gos_model

Routines Called: none

Parameters: UNS_4 *p1
 UNS_4 p2
 UNS_4 p3

Returns: none

2.8.5.7 **rcl_rtn_adrs**

The **rcl_rtn_adrs** function returns the current values of RCL addressing values, as defined in **init_addressing**.

The function call is **rcl_rtn_adrs(p1, p2, p3)**, where:

p1 is the address to return the memory location to store the RCL commands

p2 is the address to return the AAM location corresponding to the *p1* address

p3 is the address to return the number of bytes available for RCL commands

Called By: dtp_compiler
 dtp_trav1
 dtp_trav2

Routines Called: none

Parameters:	UNS_4	**p1
	UNS_4	*p2
	UNS_4	*p3

Returns:	none
----------	------

2.8.5.8 rcl_set_label

The `rcl_set_label` function is used to specify that a given label refers to the current location in active area memory (the location in `rcl_aam_adr`).

The function call is `rcl_set_label(m)`, where *m* is the label to set with the AAM location.

Called By:	double_lite dtp_trav1 dtp_trav2 replace_mod single_lite test_commands triple_lite vasi_lite
------------	--

Routines Called:	ERRMSG printf ROOM4LABEL
------------------	--------------------------------

Parameters:	UNS_4	m
-------------	-------	---

Returns:	none
----------	------

2.8.5.9 rcl_set_cntlbl

The `rcl_set_cntlbl` function identifies a section of data for output.

The function call is `rcl_set_cntlbl(m, em)`, where:

m is a previously set label that identifies the beginning of the data
em is the label associated with an output count

`rcl_set_cntlbl` stores in *em* the number of words from the address stored in *m* to the current AAM address. Output commands that refer to the set count label *em* are later patched with this data.

Called By:	double_lite single_lite test_commands
------------	---

	triple_lite	
	vsi_lite	
Routines Called:	ERRMSG	
	printf	
	ROOM4LABEL	
Parameters:	UNS_4	m
	UNS_4	cm
Returns:	none	

2.8.5.10 rcl_lblcmd

The rcl_lblcmd function generates a DTP label command.

The function call is **rcl_lblcmd(name, wd_cnt, id, rel, lbl)**, where:

name is a pointer to the name of the calling routine
wd_cnt is the total number of words the function will generate for the command
id is the command id value
rel is the relative addressing flag
lbl is the label the command branch value is associated with

rcl_lblcmd does the following:

- Uses the ROOMCHECK macro to make sure there is room for the command.
- Uses the ROOM4LABEL macro to make sure there is room for the label.
- Pushes the address and label address on the stack to patch.
- Saves the correct addressing.
- Copies the additional data.
- Updates memory data.

When rcl_lblcmd places the command location on the stack, *rel* is stored as the branch data. *rel* is set to 90 for absolute addressing, and is set to rcl_aam_adr for relative addressing. When patching occurs, this value is subtracted from the patch label to generate the relative or absolute value.

If *wd_cnt* is greater than 1, the data following *lbl* on the function stack is appended to the command.

Called By:	dtp_bcn
	dtp_bcnr
	dtp_bcz
	dtp_bczt
	dtp_bdgr
	dtp_bdlr
	dtp_bgn
	dtp_bgz

dtp_bnz
dtp_bnzc
dtp_bru
dtp_brur
dtp_brz
dtp_brzc
dtp_fov
dtp_fovr
dtp_gdc
dtp_gdcic
dtp_gdcir
dtp_gdcn
dtp_gdcnr
dtp_gdcr
dtp_lmi
dtp_lmir
dtp_lod
dtp_lodr
dtp_lwd
dtp_lwdr
dtp_osd
dtp_owd
dtp_owdsc
dtp_owr
dtp_owrsc
dtp_sub
dtp_subr
dtp_tldr
dtp_tldr
poly_efs
poly_efsr

Routines Called: rcl_push
 ROOM4LABEL
 ROOMCHECK

Parameters:	char	*name
	UNS_4	wd_cnt
	UNS_1	id
	UNS_4	rel
	UNS_4	lbl

Returns: none

2.8.5.11 rcl_command

The rcl_command function generates a DTP command with no label.

The function call is rcl_command(name, wd_cnt, id, data), where:

name is a pointer to the routine name
wd_cnt is the total number of command WORDs
id is the command id value
data is the data for the command

rcl_command does the following:

- Uses the ROOMCHECK macro to make sure there is room for the command.
- Copies the data.
- Puts the command id in memory.
- Updates memory data.

Called By:

dtp_bcnrs
dtp_bcns
dtp_bczs
dtp_bczs
dtp_bdgrs
dtp_bdlrs
dtp_bgnrs
dtp_bgzs
dtp_blm
dtp_bnzrs
dtp_bnzs
dtp_bpc
dtp_bpcx
dtp_brurs
dtp_brus
dtp_brzrs
dtp_brzs
dtp_dot
dtp_elm
dtp_end
dtp_fovrs
dtp_fovs
dtp_gdcirs
dtp_gdcis
dtp_gdcnrs
dtp_gdcns
dtp_gdcrs
dtp_gdcs
dtp_gr
dtp_lmirs
dtp_lmis
dtp_lodrs
dtp_lods
dtp_lwdrs
dtp_lwds
dtp_mml
dtp_mmpre
dtp_mmpst
dtp_mwd
dtp_ngc
dtp_oio

dtp_oos
 dtp_osds
 dtp_owds
 dtp_owo
 dtp_owrs
 dtp_rc
 dtp_subrs
 dtp_subs
 dtp_tbc
 dtp_tbdrrs
 dtp_tbrs
 poly_flu
 poly_fsw
 poly_lmf
 poly_lsc
 poly_rmmf
 poly_rm1
 poly_rm2
 poly_rm3
 poly_rm4
 poly_sm1
 poly_sm2
 poly_sm3
 poly_sm4
 poly_tog

Routines Called: ROOMCHECK

Parameters:	char	*name
	UNS_4	wd_cnt
	UNS_1	id
	UNS_4	data

Returns: none

2.8.5.12 rcl_component

The rcl_component function generates a Poly Processor component command.

The function call is **rcl_component(name, wd_cnt, incr, id, bal, lt, data)**, where:

name is a pointer to the name of the calling routine
wd_cnt is the total number of words the function will generate for the command
incr is the count increment used to initialize component data
id is the command id value
bal is the Ballistics bit
lt is the local terrain bit
data is the first piece of additional data

rcl_component does the following:

- Uses the ROOMCHECK macro to make sure there is room for the command.
- Saves the component pointers for count updates.
- Sets the component id.
- Sets the Ballistics bit if any polygons in the component need to be checked for Ballistics intersections.
- Sets the local terrain bit if any polygons in the component need to be included in the local terrain message sent to the Simulation Host.
- If *wd_cnt* is greater than 1, zeroes the second word of the component.
- Copies the additional data.
- Uses the INCR_COMPONENT macro to update the component's word count, polygon count, and vertex count in the Poly component.
- Updates memory data.

Called By: poly_bvc
 poly_gc
 poly_pc
 poly_sc
 poly_sci
 poly_sec

Routines Called: INCR_COMPONENT
 ROOMCHECK

Parameters:	char	*name
	UNS_4	wd_cnt
	UNS_4	incr
	UNS_1	id
	UNS_1	bal
	UNS_1	lt
	UNS_4	data

Returns: none

2.8.5.13 . rcl_data

The rcl_data function provides additional data for a poly component command.

The function call is **rcl_data(name, wd_cnt, incr, data)**, where:

name is the name of the calling routine

wd_cnt is the total number of words the function will generate for the command

incr is the count increment used to initialize component data

data is the first piece of additional data

rcl_data does the following:

- Uses the ROOMCHECK macro to make sure there is room for the command.
- Copies the data.

- Updates memory data.
- Uses the INCR_COMPONENT macro to update the component's word count, polygon count, and vertex count in the Poly component.

Called By: dtp_trav1
 poly_ab
 poly_inf
 poly_poly
 poly_sci
 poly_stamp
 poly_vtxe
 poly_vtxl

Routines Called: INCR_COMPONENT
 ROOMCHECK

Parameters:	char	*name
	UNS_4	wd_cnt
	UNS_4	incr
	UNS_4	data

Returns: none

2.8.5.14 rcl_stuff_data

The rcl_stuff_data function places a specified number of data words found in a specified location of user memory into successive memory locations. This function is used to add data to the processing path when no function is available to produce the desired effect.

The function call is rcl_stuff_data(cpf, wd_cnt), where:

cpf is a pointer to the data to be added
wd_cnt is the amount of data to copy

rcl_stuff_data does the following:

- Uses the ROOMCHECK macro to make sure there is room for the data.
- Copies the data.
- Updates memory data.

Called By: dtp_trav2
 poly_lmf
 poly_mmf

Routines Called: ROOMCHECK

Parameters:	UNS_4 UNS_4	*cpf wd_cnt
-------------	----------------	----------------

Returns:	none
----------	------

2.9 User Interface Mode (/cig/libsrc/libgossip)

This section describes the functions that make up the User Interface Mode (Gossip) CSC. This CSC provides a user interface that allows various debugging and query features during runtime operation. Gossip provides the ability to interrogate system performance, view and modify system memory, and debug real-time problems.

The functions available to the Gossip user include the following:

- Display data from the Ballistics database.
- Display data from the terrain and DED databases.
- Initiate Flea (the Simulation Host emulator).
- Display or modify simulation memory.
- Display static and dynamic models.
- Invoke a DTP emulator.
- Interface to the 2-D overlay processor (TX backends only).
- Display and change various system variables.
- Display the contents of message packets.
- Enable debug mode.
- Display and modify the viewport configuration tree.
- Enable and disable frame interrupts.
- Place a calibration pattern on a" channels.
- Change the default database or configuration file.
- Reset timers.

The Gossip task runs at the lowest priority, to prevent interference with the simulation.

Most Gossip actions are selected from menus. Not all Gossip menus are displayed by functions in the User Interface Mode CSC. Selection of an option from a Gossip menu may invoke a menu function in another CSC. Such menus are described in this document in the CSC to which they belong.

The CSUs contained in this CSC are identified in Figure 2-13 and described in this section.

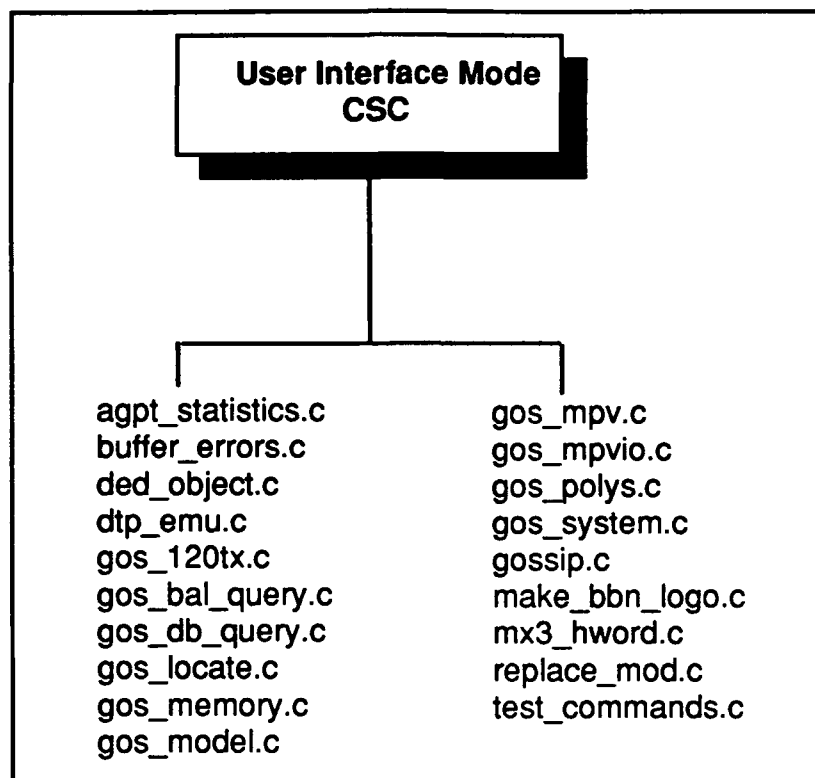


Figure 2-13. Gossip CSUs

2.9.1 agpt_statistics.c

The agpt_statistics function is not used by the standard GT100 system.

2.9.2 buffer_errors.c

The functions in the buffer_errors.c CSU are not used by the standard GT100 system.

2.9.3 ded_object.c

The functions in the ded_object.c CSU are used to work with the DED model list. These functions are:

- ded_setup
- ded_init_mdl_addr
- ded_add_model
- ded_add_effect
- ded_model_end_addr
- ded_adjust_addr_tables
- ded_uninit
- ded_sub_end
- ded_cross_border
- ded_relocate_model

- ded_process_directory
- ded_dtp_trace
- ded_load_dtp_code
- ded_model_offset
- ded_print_tables
- ded_object_debug

Note: These functions are not currently used.

2.9.3.1 ded_setup

The ded_setup function sets up the data required to work with the DED (dynamic elements database) model list.

The function call is `ded_setup(ded_start_addr, ded_size, avail_gm, mod_ad_table_P, mod_cat_table_P, spef_ad_table_P, spef_cat_table_P, total_models, total_effects, gm_end_addr, max_entries, vme_offset)`, where:

ded_start_addr is the starting location for loading dynamic models
ded_size is the amount of memory available for all dynamic models
avail_gm is the amount of space in generic memory for model information
mod_ad_table_P is a pointer to the model address table
mod_cat_table_P is a pointer to the model catalog table
spef_ad_table_P is a pointer to the special effects address table
spef_cat_table_P is a pointer to the special effects catalog table
total_models is the total number of models
total_effects is the total number of special effects
gm_end_addr is the highest address in generic memory
max_entries is the maximum number of DED entries
vme_offset is the VME offset to active area memory

ded_setup does the following:

- Allocates space for the model and special effects address and catalog tables.
- Clears the model and special effects address tables.
- Allocates space for the scratch memory address table used for models and effects.
- Calls ded_init_mdl_addr to initialize the model address array.

The function returns TRUE if successful. It returns FALSE if memory could not be allocated for one or more tables.

Called By: none

Routines Called: ded_init_mdl_addr
 malloc
 printf

Parameters:	WORD	ded_start_addr
	WORD	ded_size
	WORD	avail_gm

MODEL_TABLE_STRUCT	**mod_ad_table_P
CATALOG_TABLE_STRUCT	**mod_cat_table_P
MODEL_TABLE_STRUCT	**spef_ad_table_P
CATALOG_TABLE_STRUCT	**spef_cat_table_P
HWORD	total_models
HWORD	total_effects
WORD	gm_end_addr
HWORD	max_entries
INT_4	vme_offset

Returns: 1 (TRUE)
 0 (FALSE)

2.9.3.2 ded_init_mdl_addr

The `ded_init_mdl_addr` function initializes the model address array (`Vmdl_addr[]`). This function is called when the DED object is set up. This array is used as a working area to build the model and effect tables. The array contains one element for each model and each effect.

The function call is `ded_init_mdl_addr()`. The function always returns TRUE.

Called By: `ded_setup`

Routines Called: none

Parameters: none

Returns: 1 (TRUE)

2.9.3.3 ded_add_model

The `ded_add_model` function adds a model to the model address array, and increments the model and entry counts. The element's `model_flag` is set to TRUE.

The function call is `ded_add_model(model_index, model_addr)`, where:

model_index is the catalog index
model_addr is the model's starting address

The function always returns TRUE.

Called By: none

Routines Called: none

Parameters:	HWORD WORD	model_index model_addr
-------------	---------------	---------------------------

Returns:	1 (TRUE)
----------	----------

2.9.3.4 ded_add_effect

The `ded_add_effect` function adds a special effect to the model address array, and increments the effect and entry counters. The element's `model_flag` is set to FALSE.

The function call is `ded_add_effect(effect_index, effect_addr)`, where:

effect_index is the catalog index
effect_addr is the effect's starting address

The function always returns TRUE.

Called By:	none
------------	------

Routines Called:	none
------------------	------

Parameters:	HWORD WORD	effect_index effect_addr
-------------	---------------	-----------------------------

Returns:	1 (TRUE)
----------	----------

2.9.3.5 ded_model_end_addr

The `ded_model_end_addr` function finds the ending address of each model and effect in the model address array. It then sets the `zero_end_addr` (ending address) variable in each element in the array.

The function call is `ded_model_end_addr()`. The function always returns TRUE.

Called By:	ded_process_directory
------------	-----------------------

Routines Called:	printf	(in debug mode only)
------------------	--------	----------------------

Parameters:	none
-------------	------

Returns:	1 (TRUE)
----------	----------

2.9.3.6 ded_adjust_addr_tables

The `ded_adjust_addr_tables` function determines the AAM starting and ending address of each model and effect by offsetting them from the DED's AAM starting address. It then sets each element's `aam_start_addr`, `aam_end_addr`, and offset in the model address array. It also sets the `hw_address` for each entry in the model table and the effect table.

The function call is `ded_adjust_addr_tables(aam_offset, mdl_table_P, spef_table_P)`, where:

aam_offset is the DED's AAM start address

mdl_table_P is a pointer to the model address table

spef_table_P is a pointer to the special effects address table

The function always returns TRUE.

Called By: ded_process_directory

Routines Called: printf (in debug mode only)

Parameters:	WORD	aam_offset
	MODEL_TABLE_STRUCT	*mdl_table_P
	MODEL_TABLE_STRUCT	*spef_table_P

Returns: 1 (TRUE)

2.9.3.7 ded_uninit

The `ded_uninit` function frees the memory allocated for the model address array. This function is called when `ded_process_directory` finishes processing all models and effects.

The function call is `ded_uninit()`.

Called By: ded_process_directory

Routines Called: free

Parameters: none

Returns: none

2.9.3.8 ded_sub_end

The `ded_sub_end` function returns the ending address of the DTP subroutines as *sub_end*.

The function call is `ded_sub_end()`.

Called By: `ded_process_directory`

Routines Called: `none`

Parameters: `none`

Returns: `sub_end`

2.9.3.9 ded_cross_border

The `ded_cross_border` function determines whether two given addresses cross a quarter-megabyte boundary.

The function call is `ded_cross_border(start_addr, end_addr)`, where:

start_addr is the first address to check

end_addr is the second address to check

The function returns TRUE if the two addresses cross a quarter-meg boundary. It returns FALSE if the end address is before the start address, or if the two addresses do not cross a boundary.

Called By: `ded_process_directory`
`ded_relocate_model`

Routines Called: `none`

Parameters: `WORD` `start_addr`
`WORD` `end_addr`

Returns: `1 (TRUE)`
`0 (FALSE)`

2.9.3.10 ded_relocate_model

The `ded_relocate_model` function moves a model from a boundary crossing area to the end of available generic memory.

The function call is `ded_relocate_model(model_index, descriptor)`, where:

model_index is the index of the model to be moved

descriptor is a description to identify the model (displayed in the output message)

`ded_relocate_model` does the following:

- Makes sure there is enough generic memory available for the model.
- Makes sure the new address range does not cross a quarter-meg boundary. If it does, calls itself to try again.
- Subtracts the model size from the amount of available generic memory.
- Outputs the model's new location to stdout.

The function returns the model's new start address if successful. It returns 0 if the model size exceeds the amount of available generic memory,

Called By:	<code>ded_process_directory</code> <code>ded_relocate_model</code> (recursive)
Routines Called:	<code>ded_cross_border</code> <code>ded_relocate_model</code> <code>printf</code>
Parameters:	<code>INT_2</code> <code>char</code> <code>model_index</code> <code>descriptor[]</code>
Returns:	0 <code>dest_start</code>

2.9.3.11 ded_process_directory

The `ded_process_directory` function processes the entries in the model address array to build the model and effect tables.

The function call is `ded_process_directory mdl_table_P, sp ef_table_P, ded_fd, f_addr)`, where:

mdl_table_P is a pointer to the model address table

sp ef_table_P is a pointer to the special effects address table

ded_fd is the DED file's file descriptor

f_addr is the starting address of the model list in the DED file

ded_process_directory does the following:

- Initializes the subroutine data start address and offset to 0.
- Calls ded_sub_end to get the subroutine data end address.
- Calculates and sets the subroutine size, AAM start and end address, and offset.
- If subroutine data was detected, calls ded_cross_border to see if the data crosses a quarter-meg boundary. If so, relocates the subroutine data.
- Calls ded_model_end_addr to determine each model's end address.
- Calls ded_adjust_addr_tables to determine each model's start and end AAM addresses.
- Calls ded_load_dtp_code to load the model list into AAM.
- Adjusts the first free gm address and the amount of available gm by the number of bytes loaded by ded_load_dtp_code.
- Processes each entry in the model address array:
 - If the entry is a model, adds it to the model table and sets its description to MDL.
 - If the entry is an effect, adds it to the effect table and sets its description to SPEF.
 - Calls ded_cross_border to see if the model/effect crosses a quarter-meg boundary in AAM; if yes, calls ded_relocate_model to move the model.
 - Calls ded_dtp_trace to trace the DTP commands for the model.
 - Verifies that the model is within the area of memory allocated to the DED.
- Adjusts all the model and effect table address to VME addresses. If the model or effect is not present, fills in with default.
- If ded_debug mode is enabled, calls ded_print_tables to display the entries in the model and effect tables.
- Calls ded_uninit to free the memory allocated to the model address array.

The function returns the DED end address (Vfree_gm_addr + Vvme_offset).

Called By: none

Routines Called: ded_adjust_addr_tables
 ded_cross_border
 ded_dtp_trace
 ded_load_dtp_code
 ded_model_end_addr
 ded_print_tables
 ded_relocate_model
 ded_sub_end
 ded_uninit
 printf

Parameters:	MODEL_TABLE_STRUCT	*mdl_table_P
	MODEL_TABLE_STRUCT	*spef_table_P
	int	ded_fd
	INT_4	f_addr

Returns: Vfree_gm_addr + Vvme_offset

2.9.3.12 ded_dtp_trace

The `ded_dtp_trace` function traces the DTP commands for a particular model and adjusts the referenced addresses from generic to specific AAM addresses.

The function call is `ded_dtp_trace(model_index, descriptor, cat_index, relocate_flag)`, where:

model_index is the model's index in the model address array
descriptor is a description of the model, for display purposes
cat_index is the model's index in the catalog table
relocate_flag is **TRUE** if the model had to be relocated because it crossed a quarter-meg boundary, or **FALSE** if it did not

`ded_dtp_trace` does the following:

- Verifies that the model is in AAM.
- Traces the DTP command required to process the model.
- Makes sure the final command address is valid.

The function returns 0 if successful. It returns 1 (TRUE) if the model is detected to be outside the bounds of active area memory. It returns -1 if the final command address is out of range.

Called By: `ded_process_directory`

Routines Called: `ded_model_offset`
`printf`

Parameters:	<code>INT_2</code>	<code>model_index</code>
	<code>char</code>	<code>descriptor[]</code>
	<code>HWORD</code>	<code>cat_index</code>
	<code>BOOLEAN</code>	<code>relocate_flag</code>

Returns: `0`
`1 (TRUE)`
`-1`

2.9.3.13 ded_load_dtp_code

The `ded_load_dtp_code` function loads the model list into generic AAM, and returns the number of bytes loaded.

The function call is `ded_load_dtp_code(file_desc, file_addr, aam_addr, block_size, free_aam_size)`, where:

file_desc is the file descriptor of the DED file

file_addr is address in the file from which to start loading data
aam_addr is the address in AAM at which to start loading data
block_size is the number of bytes of data to be loaded
free_aam_size is the amount of generic memory available for the models

ded_load_dtp_code does the following:

- Adjusts the starting AAM address by the VME offset.
- Finds the beginning of the specified file.
- If *block_size* is greater than *free_aam_size* (indicating there is not enough generic memory for all DED models):
 - Outputs a message showing the needed and available sizes.
 - Reads in as many models as will fit.
- If there is enough memory, reads in the specified *block_size*.

The function returns the number of bytes loaded. This is *free_aam_size* if not enough memory was available, or *block_size* if there was enough memory for all models.

Called By: *ded_process_directory*

Routines Called: *printf*
XLSEEK
XREAD

Parameters:	<i>int</i>	<i>file_desc</i>
	<i>INT_4</i>	<i>file_addr</i>
	<i>WORD</i>	<i>aam_addr</i>
	<i>WORD</i>	<i>block_size</i>
	<i>WORD</i>	<i>free_aam_size</i>

Returns: *free_aam_size*
block_size

2.9.3.14 *ded_model_offset*

The *ded_model_offset* function finds the AAM offset of a model given a zero-based address.

The function call is *ded_model_offset(zero_addr)*, where *zero_addr* is the address to find. *ded_model_offset* does the following:

- Checks to see if the specified address is in the initial subroutine area. If so, returns the offset to the subroutine area.
- Compares the specified address to the starting and ending addresses of each model in the model address array.
- If the address is found to be between a model's starting and ending addresses, returns the offset to that model.

The function returns the offset to the subroutine data if it finds the specified address in the initial subroutine area. It returns the model's offset if it finds the specified address in the model area. It returns -1 if the specified address is not in either area.

Called By: ded_dtp_trace

Routines Called: none

Parameters: INT_4 zero_addr

Returns: Vsub_data.offset
mdl_P->offset
-1

2.9.3.15 ded_print_tables

The ded_print_tables function displays all data for all model and effect entries in the model address array. It also displays the entry's hardware address from the model or special effect table. This function is called after the model list is processed, but only if the ded_debug_flag has been enabled through Gossip.

The function call is **ded_print_tables(mdl_table_P, spef_table_P)**, where:

mdl_table_P is a pointer to the model table
spef_table_P is a pointer to the special effect table

Called By: ded_process_directory

Routines Called: printf

Parameters: MODEL_TABLE_STRUCT *mdl_table_P
MODEL_TABLE_STRUCT *spef_table_P

Returns: none

2.9.3.16 ded_object_debug

The ded_object_debug function toggles the ded_object debug flag. If this flag is enabled, descriptive information is output to stdout while the model list is processed. This function is called to enable the flag if the Gossip user selects the L ("debug DED load") option from the Database Query menu. It is called to disable the flag if the user selects the I ("don't debug DED load") option from the same menu.

The function call is `ded_object_debug(flag)`, where *flag* is **TRUE** to turn debug mode on, or **FALSE** to turn it off. The function sets the flag and returns its new state.

Called By: gos_db_query

Routines Called: none

Parameters: **BOOLEAN** **flag**

Returns: Vded_debug_flag

2.9.4 dtp_emu.c

The functions in the `dtp_emu.c` CSU are used to emulate the Data Traversal Processor (DTP) for testing and debugging. These functions are:

- dtp_emu
- display
- outdisplay
- hxflt
- hexdisplay
- ftoh
- htof
- mat_mult
- get_lm

2.9.4.1 dtp_emu

The `ntp_emu` function is a DTP emulator used in debugging. This function is invoked when the user selects the 6 ("ntp emulator") option from the Gossip main menu.

The DTP is a micro-coded processor board that sends data to the Polygon Graphics Processor, based on commands placed in active area memory by the DTP Command Generator. `dtp_emu` emulates the functions performed by the DTP.

The function call is `ntp_emu()`. Once `ntp_emu` is invoked, the Gossip user can request the following:

- Set poly data display mode on or off.
- Set the display mode to float or hex.
- Set tracing on or off.
- Set system interrupts on or off.
- Display the current emulator modes (display, poly data, system interrupt, and trace) and the DTP stack pointer.
- Display the DTP stack
- Start the DTP emulator.
- Step through the various DTP commands.
- Restart the emulator.

- Set the memory address for the emulator program counter.
- Set the address of the AAM peek (view) register.
- Set the address of the emulator peek (view) register.
- Write the contents of AAM.
- Set break points (currently not implemented).

Called By: gossip_tick

Routines Called: display
ftoh
get_lm
hexdisplay
htof
hxflt
mat_mult
printf
scanf
sqrt
sysrup_off
sysrup_on
unbf_getchar

Parameters: none

Returns: none

2.9.4.2 display

The display function is used to convert hexadecimal digits or floating point numbers for display purposes.

The function call is **display(ptr, num, poly)**, where:

ptr is a pointer to the data in AAM

num is the number of characters to convert

poly is **LOAD** if a load module is being processed, or **POLY** if a polygon is being processed

The function always returns 1.

Called By: dtp_emu

Routines Called: hxflt
printf

Parameters: INT_4 **ptr

INT_2
INT_2

num
poly

Returns: 1

2.9.4.3 outdisplay

The outdisplay function is used to display formatted data depicting polygon commands in the DTP processing path.

The function call is **outdisplay(ptr, wd_count)**, where:

ptr is the AAM pointer to the start of the Poly Processor command
wd_count is the number of bytes in the command

The function returns 0 if successful, or 1 if the command could not be displayed.

This function is not currently used.

Called By: none

Routines Called: hxflt
printf

Parameters: INT_4 **ptr
WORD wd_count

Returns: 0
1

2.9.4.4 hxflt

The hxflt function is used to convert hexadecimal characters for output to the display.

The function call is **hxflt(h)**, where *h* is the character to be converted.

Called By: display
dtp_emu
outdisplay

Routines Called: htof
printf

Parameters: WORD h

Returns: none

2.9.4.5 hexdisplay

The hexdisplay function is used to display hexadecimal numbers.

The function call is **hexdisplay(pntr, args)**, where:

pntr is the AAM address of the data to be displayed
args is the number of digits to display

Called By: dtp_emu

Routines Called: printf

Parameters: INT_4 **pntr
 INT_2 args

Returns: none

2.9.4.6 ftoh

The ftoh function is used to convert an IEEE floating point value to internal hexadecimal representation for display. It returns a pointer to the hex value.

The function call is **ftoh(f, h)**, where:

f is the floating point value
h is the hexadecimal equivalent

Called By: dtp_emu
 mat_mult

Routines Called: none

Parameters: REAL_4 *f
 WORD *h

Returns: *h

2.9.4.7 **htof**

The **htof** function is used to convert a hexadecimal number to IEEE floating point for display. It returns a pointer to the floating point value.

The function call is **htof(h, f)**, where:

h is the hexadecimal value

f is the floating point equivalent

Called By: dtp_emu
 get_lm
 hxflt
 mat_mult

Routines Called: none

Parameters: WORD *h
 REAL_4 *f

Returns: *f

2.9.4.8 **mat_mult**

The **mat_mult** function is used to multiply (concatenate) two matrices to generate a third matrix.

The function call is **mat_mult(a, b, c)**, where:

a is the address of the first matrix

b is the address of the second matrix

c is the address of the result matrix

Called By: dtp_emu

Routines Called: ftoh
 htof
 printf

Parameters: WORD *a
 WORD *b
 WORD *c

Returns: none

2.9.4.9 get_lm

The `get_lm` function is used to simulate the DTP function of getting the next load module pointer for processing.

The function call is `get_lm(flag)`, where *flag* is 0 (open -> hdglut -> lmlut), 1 (lmlut), 2 (close), or 3 (hdglut -> lmlut).

The function returns 1 if successful, or 0 if an error occurred.

Called By: dtp_emu

Routines Called: htof
printf
XCLOSE
XLSEEK
XOPEN
XREAD

Parameters: INT_2 flag

Returns: 0
1

2.9.5 gos_120tx.c

The `gos_120tx` function provides options to the Gossip user that apply to TX backends only. These options all deal with 2-D overlays, the MPV, the GSP, and the Force board. This function is invoked when the user selects the 4 ("120tx menu") option from the Gossip main menu.

The function call is `gos_120tx()`. The function does the following:

- Sets a pointer to the Force board intertask mailbox.
- Calls `bus_error` to verify that backend 0 contains a Force board; if no board is found, outputs a message telling the user to select another Force board.
- Processes the keystroke entered by the user.

The following table identifies the major steps performed by `gos_120tx` for each option on its menu. Options flagged with an asterisk (*) are supported by `gos_120tx` but do not appear on the menu.

120TX Menu Option	Processing by gos_120tx
? Display this menu	Displays valid options.
!* Enable debug mode	Set debug_enable to TRUE.
1 Start/Stop 2D updates	Toggles gsp_io_flag.
2 Enable/Disable Force Timers	(not currently implemented)
d* Display DR11 packet buffers	Calls host_if_debug.
f Select Force Board	Prompts user for Force board id (0 or 1); sets pointer to board's mailbox; calls bus_error to find board.
g Talk to 2D process/mem	Calls gos_mpv.
H MPV/Force message interface	Calls gos_mpvio.
m* Memory examine/modify	Calls gos_memory.
q* Quit	Exits.
R Reload MPV files & task	Calls mpvideo_stop to halt MPV; calls find_fn to get names of lut, data2d, and task2d files; calls mpvideo_load to load files; calls mpvideo_sim_init to restart MPV.
s* single-step mode	Calls s_step.
x exit	Exits.

If the user presses RETURN without making an entry, gos_120tx displays the following Force status variables from the Force mailbox:

- Front end control register.
- Force control register.
- Force status register.
- Force errors register.
- Force LUT wanted.
- Force LUT loaded.
- Pixel i requested.
- Pixel j requested.
- Pixel i returned.
- Pixel j returned.
- Pixel depth.
- Force timer flag.

Called By: gossip_tick

Routines Called:

- bus_error
- find_fn
- GLOB
- gos_memory
- gos_mpv
- gos_mpvio
- host_if_debug
- mpvideo_load
- mpvideo_sim_init

mpvideo_stop
printf
s_step
scanf
unbf_getchar

Parameters: none

Returns: none

2.9.6 gos_bal_query.c

The gos_bal_query function is no longer used. Previously, this function was used to display data from the Ballistics database. This process is now handled by the bx_probe function, which is part of the Ballistics Processing CSC.

2.9.7 gos_db_query.c

The functions in the gos_db_query.c CSU are used to examine database information. These functions are:

- gos_db_query
- gos_display_db_info
- gos_db_query_menu

2.9.7.1 gos_db_query

The gos_db_query function lets the Gossip user examine terrain and DED database information. This function is invoked when the user selects the 5 ("db query") option from the Gossip main menu.

The function call is **gos_db_query()**. The function does the following:

- Sets pointers to the terrain and DED tables in subsystem 0.
- Calls gos_db_query_menu to display a menu of available options and prompt the user for a selection.
- Processes the user's entries.

The following table lists the options presented on the Database Query menu and shows the major steps performed by gos_db_query to process each user selection.

Database Query Menu Option	Processing by gos_db_query
? print this menu	Calls gos_db_query_menu.
D DED data	Outputs DED file's name, version number, and size; calls gos_display_db_info.
E modify Effect	Prompts user for effect index; displays current catalog number, component count, process code, and hardware address; prompts for and sets new values.
e list effects	Displays for each effect: catalog number, component count, process code, and hardware address.
i Change DB id	Prompts user for database id (0 or 1); resets primary database control block pointer (pdbname) and all DED table pointers.
L debug DED load	Calls ded_object_debug with debug parameter set to TRUE. This causes the ded_object functions to display status messages.
l don't debug DED load	Calls ded_object_debug with debug parameter set to FALSE. This suppresses the ded_object functions from displaying status messages.
M modify model	Prompts user for model index; displays current catalog number, component count, process code, and hardware address; prompts for and sets new values.
m list models	Displays for each model: catalog number, component count, process code, and hardware address.
q quit	Exits.
T Terrain DB data	Outputs database name, version number, and size; calls gos_display_db_info.
x exit	Exits.

Called By: gossip_tick

Routines Called: blank
cup
ded_object_debug
gos_db_query_menu
gos_display_db_info
printf
scanf
unbf_getchar

Parameters: none

Returns: none

2.9.7.2 gos_display_db_info

The `gos_display_db_info` function is used by `gos_db_query` to display terrain and Dynamic Elements Database (DED) information to the Gossip user.

The function call is `gos_display_db_info(data_P)`, where `data_P` is a pointer to the database header to be displayed.

Called By: `gos_db_query`

Routines Called: `printf`

Parameters: `DB_HDR_DBASE_DATA` `*data_P`

Returns: `none`

2.9.7.3 gos_db_query_menu

The `gos_db_query_menu` function displays the Database Query menu processed by `gos_db_query`. This function is called whenever `gos_db_query` is invoked, and is called again if the user enters ? at the "Gossip DB Query>" prompt.

The function call is `gos_db_query_menu()`. The function does the following:

- Clears the screen.
- Displays the Database Query menu.
- Sets the current prompt to "Gossip DB Query>."
- Displays the "Gossip DB Query>" prompt.

The user's keystroke is processed by `gos_db_query`. For a description of the options listed on the menu, see `gos_db_query`.

Called By: `gos_db_query`

Routines Called: `blank`
`cup`
`printf`
`strcpy`

Parameters: `none`

Returns: `none`

2.9.8 gos_locate.c

The `gos_locate` function builds a hull-to-world matrix from the configuration tree's world-to-hull matrix. This function is called by `gos_model` if the user chooses to add a new model to the simulation environment.

The function call is `gos_locate(mtx_h_w)`, where `mtx_h_w` is a pointer to the hull-to-world matrix to be returned by `gos_locate`.

`gos_locate` does the following:

- Calls `vpti_get_ptr_cnode` to get a pointer to the configuration tree's root node.
- Makes sure the tree has a simulated vehicle defined.
- If more than one simulated vehicle is detected, displays a list of the vehicles and prompts the user to select one. (The ability to have multiple simulated vehicles is not currently in use.)
- Generates the hull-to-world matrix based on the vehicle's word-to-hull matrix.

The function returns the hull-to-world matrix if successful. It returns `NULL` if the configuration tree is not initialized or is empty.

Called By: `gos_model`

Routines Called: `printf`
`scanf`
`vpti_get_ptr_cnode`

Parameters: `REAL_4` `*mtx_h_w`

Returns: `mtx_h_w`
`NULL`

2.9.9 gos_memory.c

The `gos_memory` function displays relatively current data about simulation memory. This function is invoked when the user selects the `m` ("memory examine/modify") option from various Gossip menus.

The function call is `gos_memory()`. The following tables lists the options that appear on the Memory menu and shows the major steps performed by `gos_memory` to process each one. Options flagged with an asterisk are supported but do not appear on the menu.

Memory Menu Option	Processing by gos_memory
?* display this menu	Displays options.
b modify block of memory	Prompts user for address, word count and pattern; calls bus_error to determine if main or Force memory; modifies data.
c copy block of memory	Prompts user for starting source address, ending source address, and starting destination address; copies data.
d display block of memory	Prompts user for address; calls bus_error to determine if main or Force memory; displays data.
m modify memory	Prompts user for starting address; calls bus_error to determine if main or Force memory; prompts user for each new value until user quits.
q* quit	Exits.
x exit	Exits.

The macros (PRINTD4, PRINTD8, PRINTHEX4, PRINTHEX8) used by gos_memory to display data are described in Appendix B. If the display requires multiple screens, gos_memory supports paging forward and backward.

Called By: gos_120tx
 gos_model
 gos_mpv
 gos_mpvio
 gos_system
 gossip_tick

Routines Called: bus_error
 PRINTD4
 PRINTD8
 printf
 PRINTHEX4
 PRINTHEX8
 scanf
 unbf_getchar

Parameters: none

Returns: none

2.9.10 gos_model.c

The functions in the gos_model.c CSU are used to display and manipulate static and dynamic models. These functions are:

- gos_model

- `rcl_set_modloc`

2.9.10.1 `gos_model`

The `gos_model` function displays dynamic and static models. This function is invoked if the user selects the 2 ("model menu") option from the Gossip main menu.

The function call is `gos_model()`. The following table lists the options supported by `gos_model`, and shows the major steps it performs to process each one. Options flagged with an asterisk are supported but do not appear on the menu.

The table shows the options displayed and supported when debug mode is enabled (i.e., `debug_enable` is TRUE). The only options listed on the Model Selection menu if debug mode is not enabled are:

- `o` plant model in tracks
- `x` exit

Additional options that are supported if debug mode is not enabled, but do not appear on the menu, are:

- `i` read file to 500000
- `k` kludge light models
- `m` examine memory
- `?` display menu
- `!` enable debug mode

When the user adds or deletes a vehicle using this menu, `gos_model` generates a Simulation Host-type message and stores it in an area of global memory used for "pretend" vehicles. For example, a request to add a static vehicle generates a `MSG_STATICVEH_STATE` message that is stored in the `pretend_sv` location. At the end of each frame, the `pretend_veh` function (in the Real-Time Processing CSC) takes all messages from the "pretend" locations and puts them into the next incoming message buffer for processing.

Model Selection Menu Option	Processing by gos_model
?* display this menu	Displays valid options, based on current state of debug_enable.
!* enable debug mode	Sets debug_enable to TRUE.
a add static vehicle	Prompts user for model type; displays vehicle id; calls gos_locate to make hull-to-world matrix; puts MSG_STATICVEH_STATE message in pretend_sv.
b add static model @ xyz	Prompts user for model type; displays vehicle id; calls gos_locate to make hull-to-world matrix; prompts user for xyz location; puts MSG_STATICVEH_STATE message in pretend_sv.
c add dynamic model @ xyz	Prompts user for model type and xyz location; calls gos_locate to make hull-to-world matrix; puts MSG_OTHERVEH_STATE message in pretend_dv.
d delete static vehicle	Prompts user for vehicle id, model type, position; puts MSG_STATICVEH_REM message in pretend_rsv.
e display effect timing	Gives user options to add effects, display effects, change effect type, display effect timing, change effect position; puts MSG_SHOW_EFFECT message in pretend_ef.
i read file to 500000	Prompts user for file name; opens, reads, closes file.
j write file from 500000	Prompts user for file name; opens, writes to, closes file.
k kludge light models	Calls rcl_init_adrs; calls rcl_init_stack; calls replace_mod; calls rcl_patch_adrs; calls rcl_init_stack; calls test_commands.
l level of detail control	Prompts user for number of models; prompts for and sets each model's type, distance, elevation, distance between models if more than one model; calls gos_locate to make hull-to-world matrix; puts MSG_OTHERVEH_STATE message in pretend_dv; gets addresses of lod transition ranges for model; gives user option to move or rotate models, or change lod transition ranges.
m examine memory	Calls gos_memory.
n change db id	Prompts user for database id; sets database control block pointer.
o plant model in tracks	Prompts user for model type; calls gos_locate to make hull-to-world matrix; puts MSG_OTHERVEH_STATE message in pretend_dv.
q quit	Exits.
r static veh view/remove	Displays each static vehicle's id, model number, load module, and xy position; shows total count.
V display view mode	Displays current dtv_therm_word.
v FREDs view mode switch	Prompts user for new view mode; sets dtv_therm_word.
x exit	Exits.

Called By: gossip_tick

2.9.11 gos_mpv.c

The gos_mpv function provides features that may be used to test and debug the MPV board. This function is called if the user selects the g ("Talk to 2D process/mem") option from the 120TX menu. It applies to TX backends only.

The function call is gos_mpv(). The function does the following:

- Sets a pointer to the Force board's mailbox.
- Calls bus_error to locate the Force board for the currently selected backend. If the board cannot be found, prompts the user to change the Force id. (The option to select a different backend is on the 120TX menu.)
- Processes the keystroke entered by the user.

The following table identifies the action taken by gos_mpv for each option it supports. Options marked with an asterisk are supported but do not appear on the menu displayed by gos_mpv.

The CHECK_FORCE macro used by gos_mpv checks to see if the Force task is running. If so, the user is asked to retry later. (This prevents the Gossip operation from interfering with processing required for the simulation.) fe_control is the front-end control register in the Force mailbox structure; the value placed in the register tells the Force task what command to perform.

FORCE-2D Communications Menu Option	Processing by gos_mpv
?* Display this menu	Displays valid options.
0 Restart 2d processor	Calls CHECK_FORCE; sets fe_control to SUBSYS_NMI_START.
1 Reset MPV board	(not currently implemented)
4 Read Host Control	Calls CHECK_FORCE; sets fe_control to SUBSYS_READ_HCTRL.
5 Write Host Control	Calls CHECK_FORCE; sets fe_control to SUBSYS_WRITE_HCTRL.
6 Read Data	Calls CHECK_FORCE; sets fe_control to SUBSYS_READ_HDATA.
7 Write Data	Calls CHECK_FORCE; sets fe_control to SUBSYS_WRITE_HDATA.
9 Halt 2D Processor	Calls CHECK_FORCE; sets fe_control to SUBSYS_STOP.
a Set GSP addr to read/write	Prompts user for the GSP address; sets gsp_temp_addr in Force mailbox.
b Set fill mem repetitions	Prompts user for number of times to fill memory; sets fill_mem_count.
e Send mail to 2D processor	Calls CHECK_FORCE; sets fe_control to SUBSYS_MAIL_SEND.

f	Display force/2D registers	Displays current values of Front End Control Register, Force Control Register, Force Status Register, Force Errors Register, GSP Address, HWORDS count, Repeat Block Fill Count.
g	Read MPV memory	Calls CHECK_FORCE; sets fe_control to SUBSYS_READ_START.
i	Start memory fill	Calls CHECK_FORCE; sets fe_control to SUBSYS_WRITE_START; uses gsp_temp_addr, data_count, and fill_mem_count set by user.
l	Load out buf with pattern	Tells user to use MEMORY FILL option and provides address of output buffer.
m*	Memory examine/modify	Calls gos_memory.
n*	Restart 2d processor	Calls CHECK_FORCE; sets fe_control to SUBSYS_NMI_START.
o	Load output buffer (16 bits)	(not currently implemented)
p	Write to MPV memory	Calls CHECK_FORCE; sets fe_control to SUBSYS_WRITE_START.
q*	quit	Exits.
r	View input data buffer	Tells user to use MEMORY READ option and provides address of input buffer.
t	One time comm test	Calls CHECK_FORCE; sets fe_control to SUBSYS_TEST_MEM.
w	Set word count to read/write	Prompts user for word count; sets data_count.
x	exit	Exits.
y	Endless comm test	Calls CHECK_FORCE; sets fe_control to SUBSYS_TEST_MEM2.

If the user presses RETURN without typing a character, gos_mpv displays the current values of the Front End Control Register, Force Control Register, Force Status Register, Force Errors Register, GSP Address, HWORDS count, and Repeat Block Fill Count.

Called By: gos_120tx

Routines Called: bus_error
CHECK_FORCE
gos_memory
printf
scanf
unbf_getchar

Parameters: none

Returns: none

2.9.12 gos_mpvio.c

The `gos_mpvio` function provides options for testing the Force-RTSW interface messages, querying Force status, and querying MPV memory. This function is called if the Gossip user selects the **H** ("MPV/Force message interface") option from the Gossip 120TX menu.

The function call is `gos_mpvio()`. `gos_mpvio` does the following:

- Calls `mpvmsg_query_buf_addr` to get the address of the MPV query buffer (used for messages passed from `gos_mpvio` to Force).
- Calls `mpvmsg_reply_buf_addr` to get the address of the MPV reply buffer (used for messages passed from Force to `gos_mpvio`).
- Calls `mpvmsg_from_buf_addr` to get the address of the MPV from buffer (used for messages passed from Force to the MPV Interface routines).
- Displays the addresses of the query and reply buffers.
- Prompts the user to enter an option and processes the user's selection.

The following table lists the options supported by `gos_mpvio`, and shows the major steps it performs to process each one. Options flagged with an asterisk are supported but do not appear on the menu.

Message Interface - RTSW to Force Messages Menu Option	Processing by <code>gos_mpvio</code>
?* display this menu	Displays valid options.
!* set debug mode	Sets <code>debug_enable</code> to TRUE.
0 change backend number	Prompts user for new backend id; calls <code>mpvmsg_query_buf_addr</code> , <code>mpvmsg_reply_buf_addr</code> , and <code>mpvmsg_from_buf_addr</code> to get new buffer addresses; displays query and reply buffer addresses.
1 3D_LUT_DOWNLOAD	Asks if user wants to specify file to download. If yes: prompts for filename; calls <code>mpvideo_load</code> . If no: prompts for table id, channel, hex pattern; pushes <code>MSG_F0_3DLUT_DOWNLOAD</code> message on query buffer.
2 3D_LUT_SWITCH	Prompts user for table id, channel; pushes <code>MSG_F0_3DLUT_SWITCH</code> message on query buffer.
3 ALL_LUT_SWITCH	Prompts user for table id and channel for 3d lut and 2d lut; pushes <code>MSG_F0_ALLLUT_SWITCH</code> message on query buffer.
4 FINAL_LUT_DWNLD	Asks if user wants to specify file to download. If yes: prompts for filename; calls <code>mpvideo_load</code> . If no: prompts for table id, channel, entry total, entry number, hex pattern; pushes <code>MSG_F0_FINAL_LUT_DOWNLOAD</code> message on query buffer.
5 FINAL_LUT_SWITCH	Prompts user for table id, channel; pushes <code>MSG_F0_FINAL_LUT_SWITCH</code> message on query buffer.

6	MODE_SELECT	Prompts user for mode, orientation, i and j resolution, i and j offset; pushes MSG_F0_MODE_SELECT message on query buffer.
7	MPV_INIT	Pushes MSG_F0_MPV_INIT message on query buffer.
8	MPV_LUT_TYPE_REQ	Pushes MSG_F0_MPV_LUT_TYPE_REQUEST message on query buffer.
9	MPV_PEEK	Prompts user for MPV start address, byte count; pushes MSG_F0_MPV_PEEK message on query buffer.
A	MPV_POKE	Prompts user for MPV address and value to poke; pushes MSG_F0_MPV_POKE message on query buffer.
B	MPV_RESET	Pushes MSG_F0_MPV_RESET message on query buffer.
C	MPV_TASK_CTL	Prompts user for task control code and MPV address; pushes MSG_F0_MPV_TASK_CONTROL message on query buffer.
D	MPV_TEST	Pushes MSG_F0_MPV_TEST message on query buffer.
d*	display packet buffers	Calls host_if_debug.
E	MPV_WRITE	Prompts user for MPV start address, byte count, and hex pattern to write; pushes MSG_F0_MPV_WRITE message on query buffer.
F	PASS_ON	(not currently implemented)
G	PIXEL_DEPTH_REQ	Prompts user for channel, i and j pixel positions, and request id; pushes MSG_F0_PIXEL_DEPTH_REQUEST message on query buffer.
H	SET_DISPLAY	Prompts user for channel and display code; pushes MSG_F0_SET_DISPLAY message on query buffer.
I	QUERY	Prompts user for Force query code and text; pushes MSG_F0_QUERY message on query buffer.
J	TRIGGER	Pushes MSG_F0_TRIGGER message on query buffer.
K	DEBUG ENABLE	Pushes MSG_F0_DEBUG_ENABLE message on query buffer.
L	DEBUG DISABLE	Pushes MSG_F0_DEBUG_DISABLE message on query buffer.
M	MPV_POKE16	Prompts user for MPV address and value to poke; pushes MSG_F0_MPV_POKE16 message on query buffer.
m*	Memory examine/modify	Calls gos_memory.
N	FROM BUFFER DISPLAY	Peeks at and displays contents of top message in from buffer.
R	RESPONSE FRM FORCE	Peeks at and displays contents of top message in reply buffer.
s*	single-step mode	Calls s_step.
x	exit	Exits.

Called By:

gos_120tx

Routines Called:

- gets
- GLOB
- gos_memory
- host_if_debug
- mpvideo_load
- mpvmsg_from_buf_addr
- mpvmsg_query_buf_addr
- mpvmsg_reply_buf_addr
- mx3_hwcoppy
- mx3_peek
- mx3_push
- mx3_skip
- printf
- s_step
- scanf
- unbf_getchar

Parameters: none

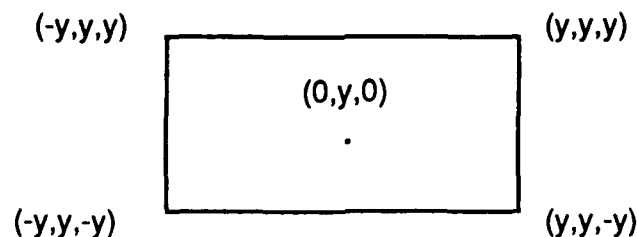
Returns: none

2.9.13 gos_polys.c

The `gos_polys` function generates monitor calibration overlays, which consist of calibration polygons, corner triangles, frame triangles, vertical and horizontal alignment bars, and colored rectangles. This function is called when the Simulation Host sends a `MSG_CALIBRATION_IMAGE` message with the image parameter set to 2 (color image). It is also called if the Gossip user selects the 2 ("color calibration pixels on") option from the Calibration menu.

The function call is `gos_polys(pcal_dat)`, where *pcal_dat* is a pointer to the calibration overlay data.

The Polygon Processor uses perspective matrices in normalized viewspace (i.e., the field-of-view is not used) when crunching on overlay polygons. The only perspective matrix required for an overlay is a matrix to swap the axes (view space into screen space). The vertices overlay can be described to the Polygon Processor as follows:



where y is the distance from the eye to the overlay.

Therefore, if the vertices of the monitor calibration overlay are given in pixel coordinates, they must be converted to the normalized viewspace coordinate system. For example, if the screen resolution is 200 x 200, a vertex with pixel coordinates (-50,100) is converted to (-1/2,1).

Called By:	cal msg_calibration_image	
Routines Called:	none	
Parameters:	CAL_OVRLY	*pcal_dat
Returns:	none	

2.9.14 gos_system.c

The `gos_system` function is used to display and change system variables. This function is called if the user selects the 3 ("system status menu") option from the Gossip main menu.

The function call is `gos_system()`. The following table lists the options supported by `gos_system` and shows the major steps it performs to process each user selection. Options flagged with an asterisk are supported but do not appear on the menu. Note the following:

- If debug mode is not enabled (i.e., `debug_enable` is FALSE), the `a` ("dtp disassembler"), `E` ("GT Hardware Menu"), and `v` ("display message block") options are not listed on the menu. The `E` option is supported, however.
- The `c` (set calibration modifier) option is not displayed on the menu if debug is enabled — it appears only if `debug_enable` is FALSE. This option is not currently implemented.

System Menu Option	Processing by gos_system
?* display this menu	Displays valid options.
!* enable debug mode	Sets debug_enable to TRUE.
2 display loc ter	Displays contents of local terrain message (header, polygon data, bounding volume data).
3 display active area data	Displays west x, east x, south y, north y, north row, east column, south row, west column, north and east pointers, south and west pointers, "cango" values.
4 display active area map	Displays center (x,y) position of each load module in AAM.
5 display load module header	Prompts user for x,y position of load module; displays load module's memory location, center, radius, direction pointers, terrain pointer, bvol pointer.
a dtp disassembler	Calls sysrup_off; prompts user for AAM PC; validates address; disassembles each DTP command.
c set calibration modifier	(not currently implemented)
d display DR11W message(s)	Calls host_if_debug.
E GT Hardware Menu	See next table.
l set display lights flags	Prompts user for hex flag mask for lights; sets display lights flags (model count, frame count, local terrain count, and overload).
m memory	Calls gos_memory.
q* quit	Exits.
s start/stop frame interrupt	Calls s_step.
v display message block	Displays message to use to use "d" option instead.
x* exit	Exits.

The following options are available if the user selects the E ("GT Hardware Menu") option from the System menu.

GT Hardware Menu Option	Processing by gos_system
?* display this menu	Displays valid options.
1 read Esifa io ports	Calls esifa_read_ports for ESIFA 0; displays video, sky, thermal, laser_set, and laser_range port values; repeats for ESIFA 1.
2 write Esifa io ports	Prompts user for ESIFA id; prompts for new port values; calls esifa_write_ports.
c write into EVC frame rate control word	Prompts user for new frame rate control word; calls write_evc_control.
d display EVC register	Calls read_evc_control; displays control word.
i initialize EVC	Calls sys_control_init.
l write into leds	Prompts user for LED value; calls set_leds.
q* quit	Exits.
r write into EVC single shot word	Prompts user for single-shot frame reset word; calls write_evc_frame.
x exit	Exits.

Called By: gossip_tick

Routines Called:

- esifa_read_ports
- esifa_write_ports
- GLOB
- gos_memory
- host_if_debug
- printf
- read_evc_control
- s_step
- scanf
- set_leds
- sys_control_init
- sysrup_off
- sysrup_on
- unbf_getchar
- write_evc_control
- write_evc_frame

Parameters: none

Returns: none

2.9.15 gossip.c

The functions in the gossip.c CSU form the main driver for the Gossip task. These functions are:

- gossip
- gossip_tick
- gos_timing_printout
- s_step
- gos_single_step
- gos_prompt
- gos_main_menu
- gos_getchar
- gos_dummy_getchar
- gos_IO_on
- gos_IO_off
- gossip_cleanup

2.9.15.1 gossip

The gossip function is the main driver for the Gossip user interface task. The gossip task is created and started at initialization time. Its priority is set lower than all other CIG tasks.

The function call is **gossip()**. When started, gossip does the following:

- Opens the console port.
- Calls gos_IO_on to enable Gossip control of the keyboard.
- Calls flea_IO_off to disable Flea control of the keyboard.
- Calls host_if_debug_init to initialize the array used to print messages every frame (if debug display is enabled).
- Calls host_enable_all_debug_msgs to enable all message types for debug display.
- Calls gos_prompt to display the "Gossip>" prompt.
- Calls poll_shutdown to see if a system shutdown has been initiated.
- Calls gossip_tick to get the user's selection.

The function exits with a 1 if it cannot open the console port.

Called By: none (task created and started at initialization time)

Routines Called:

- exit
- flea_IO_off
- gos_IO_on
- gos_prompt
- gossip_tick
- host_enable_all_debug_msgs
- host_if_debug_init
- open
- poll_shutdown
- printf
- strlen
- unbf_getchar

Parameters: none

Returns: none

2.9.15.2 gossip_tick

The `gossip_tick` function is responsible for calling the appropriate routine to process each selection made by the user on the Gossip main menu. This function is called by `gossip` when the Gossip task is started.

The function call is `gossip_tick()`. `gossip_tick` uses the `*gos_getc` function pointer to get the keystroke entered by the user, then processes it.

The following table lists the options displayed on the Gossip main menu, and shows the major steps performed by `gossip_tick` to process each user selection. The menu itself is displayed by `gos_main_menu`. Options flagged with an asterisk are supported by `gossip_tick` but do not appear on the menu.

All options are supported whether or not debug mode is enabled.

Gossip Main Menu Option	Processing by <code>gossip_tick</code>
?* display this menu	Calls <code>gos_main_menu</code>
!* enable debug	Sets <code>debug_enable</code> to TRUE.
1 calibration menu	Calls <code>cal</code> .
2 model menu	Calls <code>gos_model</code> .
3 system status menu	Calls <code>gos_system</code> .
4 120tx menu	Calls <code>gos_120tx</code> .
5 db query	Calls <code>gos_db_query</code> .
6 dtp emulator	Calls <code>dtp_emu</code> .
7 configtree menu	Calls <code>tst_tree</code> .
8 configtree init	Calls <code>vpt_trec_init</code> .
a change color config filename	Prompts user for name of color configuration file; sets <code>color_cfg_fn_to_use</code> ; sets <code>flea_color_cfg_wanted_G</code> to TRUE.
b ballistics query menu	Calls <code>bx_probe</code> .
c change configfile name	Prompts user for name of config file; sets <code>config_fn</code> .
d display interface msgs	Calls <code>host_if_debug</code> .
E display EX ethernet statistics	No effect under normal mode; calls <code>print_ex_stats</code> if running in a non-standard mode.
e change ESIFA download file	Prompts user for name of texture list file; sets <code>esifa_fn_to_use</code> ; sets <code>flea_esifa_load_wanted_G</code> to TRUE.
F change flea I/O port	Prompts user for port name for Flea console; sets <code>flea_console_port</code> .
f enter Flea menu	If flea mode is not enabled: displays error. If flea mode is enabled: calls <code>flea_IO_on</code> ; calls <code>gos_IO_off</code> ; calls <code>scratch_flea</code> ; calls <code>flea_initialized</code> ; calls <code>flea_prompt</code> ; calls <code>gos_prompt</code> .

g* <<TBD>>	Toggles transport_delay_flag.
h select host i/f msgs menu	Calls host_if_debug_menu.
i display config. messages	Toggles dr1lw_init_out.
k reset timers	Sets all mt_* timers to 0; sets force_busy_* counters to 0; sets max_out_frame counters to 0.
l list default config files	Displays names of subsystem 0's database and DED files, subsystem's 1 database and DED files, configuration file, color configuration file, and ESIFA file.
m* memory examine/modify	Calls gos_memory.
n display AGPT statistics	Calls agpt_statistics.
P PPM query	Calls gos_ppm_query.
Q Quit to GTOS	Prompts user to verify request to quit; calls rtt_shutdown.
r record i/f messages menu	Calls gos_cigsimio.
s start/stop frame interrupt	Calls s_step.
t start/stop timers	Toggles rtsw_timing_flag.
u change database name	Prompts user for subsystem id and name of database; sets db_to_use.
v change default ded name	Prompts user for subsystem id and name of DED file; sets ded_db_to_use.
w* (obsolete option for flea mode)	Treated as f ("enter Flea menu"), described above; w is supported to be backward compatible.
z* (obsolete option for flying mode)	(no longer implemented)

If the user presses RETURN without making an entry gossip_tick does the following:

- Clears the screen.
- Displays the system's version number and date, current position of the simulated vehicle, frame count, local terrain count, message count, local terrain time, rowcol_rd frame count, number of dynamic vehicles, and number of static vehicles.
- If the rtsw_timing_flag is TRUE, calls gos_timing_printout.
- Displays the configuration file name, if loaded.
- Displays the database name and the terrain address of each database partition.
- Displays the DED file name for each subsystem, if loaded.
- Displays the Ballistics file name, if loaded.
- Displays the ESIFA textures file name, if loaded.

Called By: gossip

Routines Called: *gos_getc
agpt_statistics
blank
bx_probe
cal
cup

dtp_emu
flea_initialized
flea_IO_on
flea_prompt
gos_120tx
gos_bal_query
gos_cigsimio
gos_db_query
gos_IO_off
gos_main_menu
gos_memory
gos_model
gos_ppm_query
gos_prompt
gos_system
gos_timing_printout
host_if_debug
host_if_debug_menu
isprint
isspace
printf
rtt_shutdown
s_step
scanf
scratch_flea
strcmp
tst_tree
vpt_tree_init

Parameters: none

Returns: none

2.9.15.3 gos_timing_printout

The `gos_timing_printout` function displays the latest and worst (longest) processing times for a variety of messages and frame-related times. This function is called if (1) the Gossip user presses RETURN without making an entry on the Gossip main menu, and (2) the `rtsw_timing_flag` has been enabled using the t ("start/stop timers") option on the Gossip main menu.

The function call is `gos_timing_printout()`. The display includes the following:

- Last and worst frame times.
- Last and worst packet start and end times.
- Last and worst frame interrupt times.
- Last and worst processing times for the following messages: MSG_END, MSG_OTHERVEH_STATE, MSG_SHOW_EFFECT, MSG_TRAJ_CHORD, MSG_ROUND_FIRED, MSG_STATICVEH_STATE, MSG_STATICVEH_REM, MSG_RTN_LT, MSG_PASS_ON, MSG_REQUEST_LASER_RANGE, MSG_CIG_CTL.

The timers can be reset to 0 using the k ("reset timers") option on the Gossip main menu.

Called By: gossip_tick

Routines Called: printf

Parameters: none

Returns: none

2.9.15.4 s_step

The s_step function is used to (1) enable and disable frame interrupts, and (2) enable and disable single-step mode. This function is called if the user selects the s ("start/stop frame interrupt") option from various Gossip menus.

The function call is s_step(). s_step prompts the user to set/or cancel single-step mode, then does the following:

- If the user requests "interrupts on," s_step calls sysrup_on, then sets single_step to FALSE.
- If the user requests "interrupts off," s_step calls sysrup_off, then sets single_step to FALSE.
- If the user requests "single-step mode," (used with the "display dr11 variables" option), s_step sets single_step and dr11_msg to TRUE.

Called By: gos_120tx
gos_mpvio
gos_system
gossip_tick

Routines Called: printf
sysrup_off
sysrup_on
unbf_getchar

Parameters: none

Returns: none

2.9.15.5 gos_single_step

The `gos_single_step` function forces the system to single-step a real-time frame by posting a message to the `SIMULATION_MB` mailbox. If `gos_single_step` detects that `single_step` is `TRUE`, it calls `sysrup_on`.

The function call is `gos_single_step()`.

Called By: `gos_bal_query`

Routines Called: `sysrup_on`

Parameters: `none`

Returns: `none`

2.9.15.6 gos_prompt

The `gos_prompt` function displays a greater than sign (>) followed by a text prompt. The text prompt may vary based on the current menu. The Gossip user enters the desired command next to this prompt.

The function call is `gos_prompt (newprompt)`, where *newprompt* is the text prompt to be displayed. The function also puts this value in the variable *prompt*, which is used by other Gossip functions.

Called By: `gos_main_menu`
`gossip`
`gossip_tick`
`tick`

Routines Called: `cup`
`printf`
`strcpy`

Parameters: `char` `*newprompt`

Returns: `none`

2.9.15.7 gos_main_menu

The `gos_main_menu` function displays the Gossip main menu. This menu lists the major functions available to the Gossip user. In most cases, selection of an option leads to a secondary menu. This function is called if the Gossip user enters ? at the Gossip prompt.

The function call is `gos_main_menu()`. This function clears the screen and displays the menu; it does not process the user's entry.

For a description of each item on the Gossip main menu, see `gossip_tick`.

Called By: `gossip_tick`

Routines Called: `blank`
`cup`
`gos_prompt`
`printf`

Parameters: `none`

Returns: `none`

2.9.15.8 gos_getchar

The `gos_getchar` function returns the key pressed by the user, or 0 if no key was pressed. This function is used (via the `*gos_getc` function pointer) by `gossip_tick` to determine the keystroke entered by the user on the Gossip main menu. The `*gos_getc` function pointer points to `gos_getchar` if Gossip is in control of the keyboard, and to `gos_dummy_getchar` if Flea is in control of the keyboard. `gos_IO_on` sets `*gos_getc` to `gos_getchar` when gossip starts up.

The function call is `gos_getchar()`.

Called By: `gossip_tick` (through `*gos_getc`)

Routines Called: `read_tty`

Parameters: `none`

Returns: `0`
`key`

2.9.15.9 gos_dummy_getchar

The `gos_dummy_getchar` function always returns 0. This function (instead of `gos_getchar`) is called via the `*gos_getc` function pointer while the user is in Flea mode, because Flea is responsible for handling all keystrokes. When it receives a return value of 0, `gossip_tick` assumes no key was pressed and therefore does not process the keystroke. `gos_IO_off` sets `*gos_getc` to `gos_dummy_getchar` if the user selects Flea mode.

The function call is `gos_dummy_getchar()`.

Called By: `gossip_tick` (through `*gos_getc`)

Routines Called: none

Parameters: none

Returns: 0

2.9.15.10 gos_IO_on

The `gos_IO_on` function sets the `*gos_getc` function pointer to `gos_getchar`. This gives `gossip_tick` the ability to respond to the user's keystrokes on the Gossip main menu. This function is called when gossip starts up.

The function call is `gos_IO_on()`.

Called By: `gossip tick`

Routines Called: `printf` (in debug mode only)

Parameters: none

Returns: none

2.9.15.11 gos_IO_off

The `gos_IO_off` function sets the `*gos_getc` function pointer to `gos_dummy_getchar`. This function is called if the Gossip user selects Flea mode. It has the effect of masking the user's keystrokes from `gossip_tick`, so they can be handled by Flea.

The function call is `gos_IO_off()`.

Called By: gossip_tick

Routines Called: printf (in debug mode only)

Parameters: none

Returns: none

2.9.15.12 gossip_cleanup

The gossip_cleanup function deallocates the resources owned by the gossip task. This function is called if the Gossip user requests a system shutdown by selecting the Q ("Quit to GTOS") option on the Gossip main menu. This function is called via the *task_cleanup function pointer, which points to the cleanup routine's name in the task table.

The function call is **gossip_cleanup()**.

The function returns 1 if successful, or 0 if an error occurred.

Note: This function is not yet implemented. At the current time, it simply returns a 1 if called.

Called By: poll_shutdown (through *task_cleanup)

Routines Called: none

Parameters: none

Returns: 0
 1

2.9.16 make_bbn_logo.c

The make_bbn_logo function generates a BBN logo image using polygons, frame triangles, corner triangles, and colored rectangles.

This function is called if the Simulation Host sends a MSG_CALIBRATION_IMAGE message with the image parameter set to 3 (BBN logo). It is also called if the Gossip user selects the 3 ("bbn logo on") option from the Calibration menu.

The function call is **make_bbn_logo(pcal_dat)**, where *pcal_dat* is a pointer to the image data.

Called By:	cal msg_calibration_image	
Routines Called:	none	
Parameters:	CAL_OVRLY	*pcal_dat
Returns:	none	

2.9.17 **mx3_hword.c**

The functions in the `mx3_hword.c` CSU are used by `gos_mpvio` to communicate with the MPV via the Force board. The interface uses half-word message exchanges. These functions are:

- `mx3_open`
- `mx3_push`
- `mx3_peek`
- `mx3_skip`
- `mx3_error`
- `mx3_hwcopy`

Messages sent from `gos_mpvio` to Force are prefixed with `MSG_F0`. Messages returned from Force are prefixed with `MSG_F1`. The message buffers used by `gos_mpvio` to communicate with the Force task are the following:

mpvio_query_buf (outgoing)

Used for messages sent from `gos_mpvio` to Force. These messages result from changes requested by the Gossip user.

mpvio_reply_buf (incoming)

Used for response messages returned by Force to `gos_mpvio`.

During a simulation, the real-time software communicates with the MPV using the MPV Interface routines. The message buffers and routines used by the MPV Interface functions to communicate with the Force board are different from those used by `gos_mpvio`.

2.9.17.1 **mx3_open**

The `mx3_open` function initializes a message buffer given its start address and size.

The function call is `mx3_open(dev_P, device_size)`, where:

dev_P is a pointer to the MX device (message buffer)
device_size is the size of the message buffer

The function always returns `MX_DEVICE_OPENED`.

This function is not currently used — the message buffers are opened by Force Processing functions.

Called By:	none	
Routines Called:	sc_lock sc_unlock	
Parameters:	MX2_DEVICE INT_4	*dev_P device_size
Returns:	MX_DEVICE_OPENED	

2.9.17.2 mx3_push

The `mx3_push` function pushes a message onto the message buffer. This function is used by `gos_mpvio` to pass messages to the Force board.

The function call is `mx3_push(dev_P, source_address, message_code, message_size)`, where:

dev_P is a pointer to the message buffer
source_address is the address of the message
message_code is the type of message
message_size is the number of bytes in the message

`mx3_push` does the following:

- Locks the buffer.
- Verifies that there is room in the buffer for the message.
- Copies the message to the end of the buffer.
- Unlocks the buffer.

The function returns `MX_MESSAGE_PUSHED` if successful. It returns `MX_DEVICE_FULL` if the specified message buffer is already full.

Called By:	gos_mpvio	
Routines Called:	mx3_hwcopy sc_lock sc_unlock	
Parameters:	MX2_DEVICE WORD HWORD HWORD	*dev_P source_address message_code message_size

Returns: MX_MESSAGE_PUSHED
 MX_DEVICE_FULL

2.9.17.3 mx3_peek

The `mx3_peek` function previews the message at the head of a specified buffer. This function used to determine what type of message is in the incoming buffer.

The function call is `mx3_peek(dev_P, message_code, message_size, message_addr)`, where:

dev_P is a pointer to the message buffer
message_code is the message type
message_size is the size of the message in bytes
message_addr is a pointer to the message's address

`mx3_peek` does the following:

- Locks the buffer.
- Checks to see if the specified buffer is empty.
- Sets a pointer to the first message in the buffer.
- Places the message's type and size in *message_code* and *message_size*.
- If the message code is `MX_SKIP`, starts over with the next message in the buffer.
- Places a pointer to the message in *message_addr*.
- Unlocks the buffer.

The function returns `MX_MESSAGE_PREVIEWED` if successful. It returns `MX_DEVICE_EMPTY` if the specified buffer contains no messages.

Called By: gos_mpvio

Routines Called: sc_lock
 sc_unlock

Parameters:	MX2_DEVICE	*dev_P
	WORD	*message_code
	WORD	*message_size
	BYTE	**message_addr

Returns: MX_MESSAGE_PREVIEWED
 MX_DEVICE_EMPTY

2.9.17.4 mx3_skip

The `mx3_skip` function skips over a message in the buffer. The message at the head of the buffer is flushed, and the next message moves to the top of the buffer. This function is used to remove messages from a buffer after they have been previewed and processed.

The function call is `mx3_skip(dev_P)`, where *dev_P* is a pointer to the buffer.

The function returns `MX_MESSAGE_SKIPPED` if successful. It returns `MX_DEVICE_EMPTY` if the specified message buffer contains no messages.

Called By:	gos_mpvio	
Routines Called:	sc_lock sc_unlock	
Parameters:	MX2_DEVICE	*dev_P
Returns:	MX_MESSAGE_SKIPPED MX_DEVICE_EMPTY	

2.9.17.5 mx3_error

The `mx3_error` function returns a text message for output to the operator. Messages are provided for errors as well as for normal processing states.

The function call is `mx3_error(status)`, where *status* is the current MX state.

This function is not currently used.

Called By:	none	
Routines Called:	none	
Parameters:	WORD	status
Returns:	"DEVICE CLOSED" "DEVICE TABLE FULL" "DEVICE OPENED" "DEVICE BUSY" "DEVICE EMPTY" "DEVICE FULL" "MESSAGE PUSHED"	

"MESSAGE POPPED"
 "MESSAGE PREVIEWED"
 "MESSAGE SKIPPED"
 "UNDEFINED ERROR"
 "UNDEFINED RETURN"

2.9.17.6 mx3_hwcopy

The mx3_hwcopy function performs a half-word block copy.

The function call is **mx3_hwcopy(source_P, destination_P, byte_count)**, where:

source_P is a pointer to the source data
destination_P is a pointer to the destination location
byte_count is the number of bytes to be copied

Called By: gos_mpvio
 mx3_push

Routines Called: none

Parameters:	INT_2	*source_P
	INT_2	*destination_P
	INT_2	byte_count

Returns: none

2.9.18 replace_mod.c

The functions in the replace_mod.c CSU are used to replace light models. These functions are:

- replace_mod
- single_lite
- double_lite
- triple_lite
- vasi_lite
- outahere

2.9.18.1 replace_mod

The replace_mod function puts light light models in place of others. This function is called if the Gossip user selects the **k** ("kludge light models") option from the Model Selection menu. This option appears on the menu only if debug is enabled.

The function call is **replace_mod()**. The function calls various DTP macros to generate the following light models:

- VASI Bar 1 and 2
- Hazard beacon
- Military beacon
- Commercial beacon
- White approach
- Blue taxiway
- Hazard light
- Green red runway
- White amber runway
- Runway strobes

Called By: gos_model

Routines Called:

- double_lite
- dtp_bdlr
- dtp_bru
- dtp_brz
- dtp_dot
- dtp_fov
- dtp_gr
- dtp_lod
- dtp_owd
- dtp_owo
- dtp_rc
- dtp_sub
- dtp_tpc
- dtp_tldr
- dtp_tldr
- outahere
- poly_mmf
- poly_rm1
- printf
- rcl_patch_adrs
- rcl_set_errptr
- rcl_set_label
- rcl_set_modloc
- single_lite
- triple_lite
- vasi_lite

Parameters: none

Returns: none

2.9.18.2 single_lite

The single_lite function is used to generate some of the light models.

The function call is **single_lite(lstart, hwdth, hght, piw1, piw2)**, where:

lstart is the label to set with the AAM location
hwdth is the width of the light
hght is the height of the light
piw1 is poly info word 1
piw2 is poly info word 2

Called By: replace_mod

Routines Called: poly_inf
poly_pc
poly_vtxe
poly_vtxl
rcl_set_cntlbl
rcl_set_label

Parameters:	WORD	lstart
	REAL_4	hwdth
	REAL_4	hght
	WORD	piw1
	WORD	piw2

Returns: none

2.9.18.3 double_lite

The double_lite function is used to generate some of the light models.

The function call is **double_lite(lstart, hwdth, hght, piw1, piw2)**, where:

lstart is the label to set with the AAM location
hwdth is the width of the light
hght is the height of the light
piw1 is poly info word 1
piw2 is poly info word 2

Called By: replace_mod

Routines Called: poly_inf
poly_pc
poly_vtxe
poly_vtxl
rcl_set_cntlbl
rcl_set_label

Parameters:	WORD	lstart
	REAL_4	hwdth
	REAL_4	hght
	WORD	piw1
	WORD	piw2

Returns: none

2.9.18.4 triple_lite

The triple_lite function is used to generate some of the light models.

The function call is **triple_lite(lstart, hwdth, hght, piw1, piw2, piw3)**, where:

lstart is the label to set with the AAM location
hwdth is the width of the light
hght is the height of the light
piw1 is poly info word 1
piw2 is poly info word 2
piw3 is poly info word 3

Called By: replace_mod

Routines Called:

- poly_inf
- poly_pc
- poly_vtxe
- poly_vtxl
- rcl_set_cntlbl
- rcl_set_label

Parameters:	WORD	lstart
	REAL_4	hwdth
	REAL_4	hght
	WORD	piw1
	WORD	piw2
	WORD	piw3

Returns: none

2.9.18.5 vasi_lite

The vasi_lite function is used to generate some of the light models.

The function call is **vasi_lite(lstart, hwdth, phght, hght, piw1, piw2)**, where:

lstart is the label to set with the AAM location
hwdth is the width of the VASl

phgt is <<TBD>> of the height of the VASI
hgt is the height of the VASI
piw1 is poly info word 1
piw2 is poly info word 2

Called By: replace_mod

Routines Called: poly_inf
poly_pc
poly_vtxe
poly_vtxl
rcl_set_cntlbl
rcl_set_label

Parameters:	WORD	lstart;
	REAL_4	hwdth
	REAL_4	phgt
	REAL_4	hgt
	WORD	piw1
	WORD	piw2

Returns: none

2.9.18.6 outahere

The outahere function is used to output high- or low-intensity data.

The function call is **outahere(lstart)**, where *lstart* is the label to set with the AAM location.

Called By: replace_mod

Routines Called: dtp_gr
dtp_owd
dtp_owdsc

Parameters:	WORD	lstart
-------------	------	--------

Returns: none

2.9.19 test_commands.c

The test_commands function tests each DTP command. This function is called if the Gossip user selects the k ("kludge light models") option from the Model Selection menu. This option appears on the menu only if debug is enabled.

The function call is test_commands().

Called By: gos_model

Routines Called:

- dtp_bcn
- dtp_bcnr
- dtp_bcz
- dtp_bczr
- dtp_bdgr
- dtp_bdlr
- dtp_bgn
- dtp_bgz
- dtp_bnz
- dtp_bnzr
- dtp_bru
- dtp_brur
- dtp_brz
- dtp_brzr
- dtp_dot
- dtp_elm
- dtp_end
- dtp_fov
- dtp_fovr
- dtp_gdci
- dtp_gdcr
- dtp_gr
- dtp_lmi
- dtp_lmir
- dtp_lod
- dtp_lodr
- dtp_lwd
- dtp_lwdr
- dtp_ngc
- dtp_oio
- dtp_oos
- dtp_owd
- dtp_owdsc
- dtp_owo
- dtp_owr
- dtp_rc
- dtp_sub
- dtp_subr
- dtp_tbc
- dtp_tbdr

dtp_tbr
poly_flu
poly_inf
poly_mmf
poly_pc
poly_rm1
poly_vtce
poly_vtxl
printf
rcl_patch_adrs
rcl_set_cntlbl
rcl_set_label

Parameters: none

Returns: none

2.10 Host Interface Manager (/cig/libsrc/libhost)

The Host Interface Manager CSC is responsible for exchanging data packets between the CIG and the Simulation Host via all supported physical interfaces. The interfaces currently supported are the following:

- Digital Equipment Corporation DR11-W
- Ethernet IEEE 4.02
- Ready Systems MPV
- SCSI
- Socket

For each physical interface type, the following functions are provided:

init <interface> interface

Establishes the interface by setting the *exchange_data and *exchange_data_sim function pointers to the appropriate routines. These function pointers are used by the real-time software to call the routines; the real-time functions do not know which physical interface is implemented.

open <interface> interface

Opens the interface, usually by opening the communications device as a file (host_fd). The exchange routines can then use IFX calls to read from and write to the device as if it were a file.

exchange <interface> _data

Exchanges incoming and outgoing message packets with the Simulation Host (by reading from and writing to the host_fd file) during any CIG state other than simulation.

exchange <interface> _data_sim

Exchanges incoming and outgoing message packets with the Simulation Host (by reading from and writing to the host_fd file) during a simulation.

The physical interface in use at a specific site is specified on the command line at startup. The initialize function (in the Real-Time Processing CSC) parses the command line and calls the appropriate init_interface function to establish the correct interface.

The Host Interface Manager also handles the exchange of message packets between the real-time software and Flea, the Simulation Host emulator used for stand-alone operation and testing. While running under Flea control, the CIG and Flea can exchange packets without using the physical interface routines. (By posting a message to an intertask mailbox, each process alerts the other that it has placed a packet in the INBUF or OUTBUF buffer.) Alternatively, the the system can be configured to use the DR11-W, Ethernet, or SCSI interface while running a Flea exercise.

Other functions in the Host Interface Manager CSC are used to display the contents of messages for debugging purposes. Any or all message types can be enabled for display. Through Gossip, the user can access the Host Interface Debug menu to review or change the message types that are currently enabled. Each frame, enabled messages are printed to stdout (using the print_msg_* functions in the Message Processing CSC) if the dr11w_init_out debug flag has been enabled through Gossip.

Figure 2-14 identifies the CSUs in the Host Interface Manager CSC. The functions performed by these CSUs are described in this section.

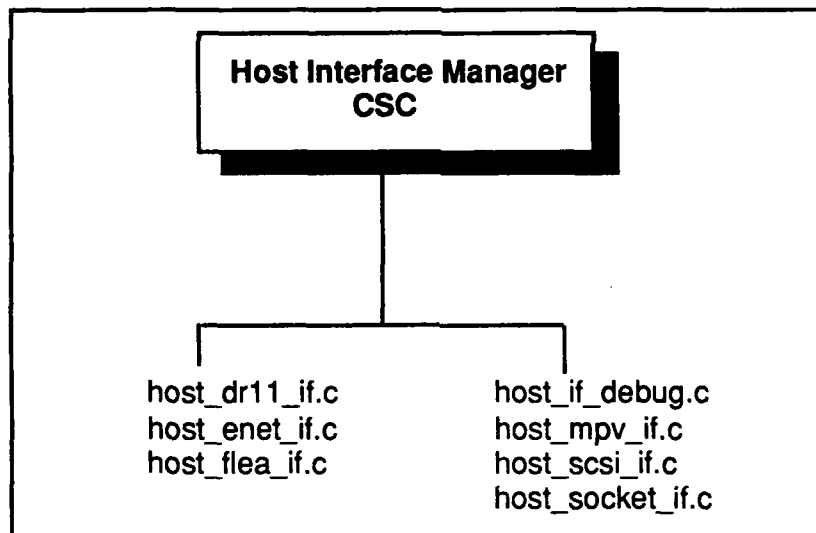


Figure 2-14. Host Interface Manager CSUs

2.10.1 `host_dr11_if.c`

The functions in the `host_dr11_if.c` CSU are used to exchange data packets over a DR11-W physical interface. These functions are:

- `open_dr11_interface`
- `exchange_dr11_data`
- `exchange_dr11_data_sim`
- `init_dr11_interface`

2.10.1.1 `open_dr11_interface`

The `open_dr11_interface` function opens the `dr0:` device as the `host_fd` file. This function is called if DR11 mode has been established. It is also called if the system is running under Flea and either DR11 or MPV mode was selected.

The function call is `open_dr11_interface()`.

If the `dr0:` device cannot be opened, `open_dr11_interface` outputs an error and exits with a 1.

Called By: `init_dr11_interface`
`open_flea_interface`

Routines Called: `exit`

open
printf

Parameters: none

Returns: none

2.10.1.2 exchange_dr11_data

The `exchange_dr11_data` function exchanges output and input buffers with the Simulation Host during any CIG state other than simulation. This function is called via the `*exchange_data` function pointer if DR11 mode has been established. It is also called via the `*exchange` function pointer if the system is running under Flea and either DR11 or MPV mode has been selected.

The function call is `exchange_dr11_data(state)`, where *state* is the current state of the CIG. `exchange_dr11_data` does the following:

- If the `dr11_pkt_debug` switch is enabled, outputs the packet sizes to stdout.
- Generates a `MSG_SYS_ERROR` message to report errors in the previous frame.
- Adds a `MSG_END` message to the outgoing packet.
- Sets the outgoing packet size.
- Calls `ifx_write` to write the packet to the `host_fd` file.
- Calls `ifx_read` to read the new incoming packet from the `host_fd` file.
- Resets the INBUF and OUTBUF pointers.
- If the `dr11w_init_out` debug switch is enabled, calls `host_if_debug` to display messages contents (if the debug display feature is enabled).

Called By:	<code>cig_config</code>	(through <code>*exchange_data</code>)
	<code>db_mcc_setup</code>	(through <code>*exchange_data</code>)
	<code>file_control</code>	(through <code>*exchange_data</code>)
	<code>flea_host_if</code>	(through <code>*exchange</code>)
	<code>get_msg_2d</code>	(through <code>*exchange_data</code>)
	<code>hw_test</code>	(through <code>*exchange_data</code>)
	<code>upstart</code>	(through <code>*exchange_data</code>)

Routines Called: `host_if_debug`
`ifx_read`
`ifx_write`
`printf`

Parameters: `INT_4` `state`

Returns: none

2.10.1.3 exchange_dr11_data_sim

The `exchange_dr11_data_sim` function exchanges output and input buffers with the Simulation Host during a simulation. This function is called (via the `*exchange_data_sim` function pointer) at the end of every frame if DR11 mode has been established.

The function call is `exchange_dr11_data_sim(state)`, where *state* is the current state of the CIG. `exchange_dr11_data_sim` does the following:

- If the `dr11_pkt_debug` switch is enabled, outputs the packet sizes to stdout.
- Calls `loc_ter_msg` to generate a local terrain message if one is required this frame.
- Generates a `MSG_SYS_ERROR` message to report errors in the previous frame.
- Adds a `MSG_END` message to the outgoing packet.
- Sets the outgoing packet size.
- Calls `ifx_write` to write the packet to the `host_fd` file.
- Calls `ifx_read` to read the new incoming packet from the `host_fd` file.
- Resets the `INBUF` and `OUTBUF` pointers.

Called By: `_set_up_for_next_frame` (through `*exchange_data_sim`)

Routines Called: `ifx_read`
`ifx_write`
`loc_ter_msg`
`printf`

Parameters: `INT_4` `state`

Returns: `none`

2.10.1.4 init_dr11_interface

The `init_dr11_interface` function establishes the DR11 as the current interface. This function is called if no physical interface was specified at startup. (DR11 is the default interface.)

The function call is `init_dr11_interface()`. The function does the following:

- Calls `open_dr11_interface` to open the DR11 device.
- Sets the `*exchange_data` function pointer to `exchange_dr11_data`.
- Sets the `*exchange_data_sim` function pointer to `exchange_dr11_data_sim`.

Called By: `initialize`

Routines Called: `open_dr11_interface`
`printf`

Parameters: none

Returns: none

2.10.2 host_enet_if.c

The functions in the `host_enet_if.c` CSU are used to exchange data packets over an Ethernet physical interface. These functions are:

- `open_enet_interface`
- `exchange_enet_data`
- `exchange_enet_data_sim`
- `slave_cig_enet_sync`
- `init_enet_interface`

Ethernet mode is selected by specifying the "e" argument on the startup command line. The operator also specifies:

- The CIG's Ethernet mode: 1 (Master CIG) or 2 (Slave CIG).
- The Simulation Host's Ethernet address.

All packet buffers include a standard Ethernet preamble which specifies the source Ethernet address, destination Ethernet address, and the protocol

2.10.2.1 open_enet_interface

The `open_enet_interface` function opens the `enet:` device as the `host_fd` file. This function is called if Ethernet mode has been established. It is also called if the system is running under Flea and Ethernet mode was selected.

The function call is `open_enet_interface()`. The function does the following:

- Opens the Ethernet device as `host_fd` in read-write mode.
- Inserts the destination (Simulation Host's) address into the preamble in the incoming Ethernet buffer. (This is required because FLEA uses INBUF as the output buffer.)
- Inserts the destination (Simulation Host's) address into the preamble in the outgoing Ethernet buffer.
- Sets the protocol in the preamble to `PROTOCOL_SIM`.
- Uses `ifx_ioctl` to set up the function pointers used by the other Ethernet interface routines to read and write to the `host_fd` quickly (bypassing IFX overhead). These function pointers are `*eccb`, `*eread`, and `*ewrite`.
- Sets `slave_init_mode` to `FALSE`.
- Sets `startup` to `TRUE`.

If the `enet:` device cannot be opened, `open_enet_interface` outputs an error and exits with a 1.

Called By: init_enet_interface
 open_flea_interface

Routines Called: bcopy
 exit
 ifx_ioctl
 ifx_open
 printf

Parameters: none

Returns: none

2.10.2.2 exchange_enet_data

The `exchange_enet_data` function exchanges output and input buffers with the Simulation Host during any CIG state other than simulation. This function is called via the `*exchange_data` function pointer if Ethernet mode has been established. It is also called via the `*exchange` function pointer if the system is running under Flea and Ethernet mode was selected.

The function call is `exchange_enet_data(state)`, where *state* is the current state of the CIG. `exchange_enet_data` does the following:

- If running in Flea mode:
 - Puts the Host's Ethernet address in the outgoing Ethernet packet.
 - If `flea_init_slave_cig_G` is TRUE:
 - * Sets the variable to FALSE.
 - * Copies INBUF to the outgoing (Flea->CIG) Ethernet buffer.
 - * Uses the `*ewrite` function pointer to write the outgoing packet to the Ethernet device.
 - Uses the `*eread` function pointer to find the packet from the CIG.
 - Copies the incoming (CIG->Flea) packet to OUTBUF.
 - Copies INBUF to the outgoing (Flea->CIG) Ethernet packet.
 - Uses the `*ewrite` function pointer to write the outgoing packet to the Ethernet device.
- If not running in Flea mode:
 - If the CIG is either a Master CIG or a Slave CIG that has started its initialization sequence:
 - * Calls `syserr` to generate a `MSG_SYS_ERROR` message.
 - * Adds a `MSG_END` message to the outgoing packet.
 - * Sets the outgoing packet size.
 - * Copies the message packet to the outgoing Ethernet buffer.
 - * Uses the `*ewrite` function pointer to write the outgoing buffer to the Ethernet device.
 - Uses the `*eread` function pointer to read the new incoming packet from the Ethernet device.
 - If startup is TRUE (indicating that this is the first write/read attempt):
 - * If the incoming buffer is empty, delays and tries again.

- * If a packet is returned, sets startup to FALSE. If the CIG is running in Slave mode, sets slave_init_mode to TRUE.
- Copies the incoming packet to INBUF.
- Resets the INBUF and OUTBUF pointers.
- If the dr11w_init_out debug switch is enabled, calls host_if_debug to display messages contents (if the debug display feature is enabled).

The function exits with a 1 if it encounters an error writing to the Ethernet device.

Called By:	cig_config	(through *exchange_data)
	db_mcc_setup	(through *exchange_data)
	file_control	(through *exchange_data)
	flea_host_if	(through *exchange_data)
	get_msg_2d	(through *exchange_data)
	hw_test	(through *exchange_data)
	upstart	(through *exchange_data)

Routines Called:	*eread	
	*ewrite	
	bcopy	
	exit	
	host_if_debug	
	max	
	printf	
	putchar	(in debug mode only)
	sc_delay	
	syserr	

Parameters:	INT_4	state
-------------	-------	-------

Returns:	none
----------	------

2.10.2.3 exchange_enet_data_sim

The exchange_enet_data_sim function exchanges output and input buffers with the Simulation Host during a simulation. This function is called at the end of every frame if Ethernet mode has been established.

The function call is **exchange_enet_data_sim(state)**, where *state* is the current state of the CIG. exchange_enet_data_sim does the following:

- If the CIG is running in Master mode:
 - Calls loc_ter_msg to generate a local terrain message if required this frame.
 - Calls syserr to generate a MSG_SYS_ERROR error message.
 - Adds a MSG_END message to the outgoing packet.
 - Sets the outgoing packet and buffer size.
 - Copies the packet from OUTBUF to the Ethernet buffer.
 - Uses the *ewrite function pointer to write the outgoing buffer to the Ethernet device.

- Starts a timer.
- Uses the **eread* function pointer to read the incoming message packet.
- If buffer cannot be read (i.e., remains empty) before the *READ_TIMEOUT* occurs, outputs an error to *stdout*.
- If a packet is found, copies it to *INBUF*.
- Resets the *INBUF* and *OUTBUF* pointers.

Called By: `_set_up_for_next_frame` (through **exchange_data_sim*)

Routines Called: **eread*
**ewrite*
bcopy
loc_ter_msg
max
printf
read_watch
start_watch
syserr

Parameters: `INT_4` `state`

Returns: `none`

2.10.2.4 `slave_cig_enet_sync`

The `slave_cig_enet_sync` function is responsible for synchronizing with another CIG.

The function call is `slave_cig_enet_sync(cig_type)`, where *cig_type* is 1 (*ENET_MASTER_CIG*) or 2 (*ENET_SLAVE_CIG*). `slave_cig_enet_sync` does the following:

- Creates an outgoing packet that contains only a *MSG_END* message.
- Copies the packet from *OUTBUF* to the outgoing Ethernet buffer.
- Puts the Simulation Host's address in the outgoing Ethernet buffer.
- If the CIG is running in Slave mode, uses the **ewrite* function pointer to write the outgoing buffer to the Ethernet device.
- Using the **eread* function pointer, tries to get the incoming buffer from the Ethernet device; tries again if the device is empty.
- Uses the **ewrite* function pointer to write the outgoing buffer to the Ethernet device.
- Resets the *INBUF* and *OUTBUF* pointers.

The function exits with a 1 if it encounters an error writing to or reading from the Ethernet device.

This function is not currently used.

Called By: `none`

Routines Called: *eread
 *ewrite
 bcopy
 exit
 max
 printf
 putchar
 sc_delay

Parameters: WORD cig_type

Returns: none

2.10.2.5 init_enet_interface

The `init_enet_interface` function establishes Ethernet as the current interface. This function is called if Ethernet mode was specified at startup.

The function call is `init_enet_interface(master_or_slave, address)`, where:

master_or_slave is 1 (ENET_MASTER_CIG) or 2 (ENET_SLAVE_CIG)
address is the Simulation Host's Ethernet address

Both the Master/Slave variable and the Simulation Host's address are specified on the startup command line.

`init_enet_interface` does the following:

- Sets `enet_mode_G` to the value specified in *master_or_slave*; this variable is used by the other Ethernet interface routines.
- Outputs the Simulation Host's Ethernet address to stdout.
- Calls `open_enet_interface` to open the Ethernet device and put the Host's address in the Ethernet buffers.
- Sets the `*exchange_data` function pointer to `exchange_enet_data`.
- Sets the `*exchange_data_sim` function pointer to `exchange_enet_data_sim`.

Called By: initialize

Routines Called: open_enet_interface
 printf
 scanf

Parameters: INT_4 master_or_slave
 char *address

Returns: none

2.10.3 host_flea_if.c

The functions in the host_flea_if.c CSU exchange data packets between the CIG and the Flea task. Flea emulates a Simulation Host, creating message packets based on input from the Gossip user "driving" the simulation vehicle via the keyboard. Flea allows a CIG to operate in stand-alone mode, without connection to a Simulation Host.

The functions in this CSU are:

- open_flea_interface
- exchange_flea_data
- flea_host_if
- init_flea_interface

Flea mode is specified by entering the "f" switch on the startup command line. The operator must also specify the desired Flea mode, which controls the interface routines used to communicate between the real-time software and Flea. Available modes are:

- 1 Normal
- 2 DR0: (DR11)
- 3 ENET: (Ethernet)
- 4 MPV
- 5 SCSI

In Normal mode, no physical interface routines are used — packets are not written to or read from a communications device. When Flea (acting as the Simulation Host) puts a new packet in the INBUF buffer, it posts a message to an intertask mailbox to alert the real-time software. When the real-time software puts a new packet in the OUTBUF buffer, it posts a mailbox message to alert Flea. Each task then gets its new packet directly from the buffer.

If DR11 or MPV mode is selected, the DR11 routines are used. Separate routines are provided for Ethernet and SCSI modes.

2.10.3.1 open_flea_interface

The open_flea_interface function calls the applicable open_interface function to set up the Flea communications path. The interface is selected based on the Flea mode entered on the startup command line.

The function call is **open_flea_interface()**. The function does the following:

- Calls the appropriate open_interface routine based on the flea_mode_G variable (set by init_flea_interface):

FLEA_DR11	Calls open_dr11_interface
FLEA_MPV	Calls open_dr11_interface
FLEA_ENET	Calls open_enet_interface
FLEA_SCSE	Calls open_scsi_interface
FLEA_NORMAL	No action
- Creates an outgoing message packet containing a MSG_END message.
- Sets the message packet size.

- Resets the INBUF and OUTBUF pointers.

Called By: init_flea_interface

Routines Called: open_dr11_interface
 open_enet_interface
 open_scsi_interface

Parameters: none

Returns: none

2.10.3.2 exchange_flea_data

The exchange_flea_data function exchanges output and input buffers with the Flea task. This function is called via the *exchange_data or *exchange_data_sim function pointer if Flea mode has been established.

The function call is exchange_flea_data(state), where *state* is the current state of the CIG. exchange_flea_data does the following:

- If the dr11_pkt_debug switch is enabled, outputs the packet sizes to stdout.
- Calls loc_ter_msg to generate a local terrain message if required this frame.
- Calls SYSERR to generate a MSG_SYS_ERROR message to report errors in the previous frame.
- Adds a MSG_END message to the outgoing packet.
- Sets the outgoing packet size.
- Posts a message to the FLEA_OUTPUT_MB mailbox.
- Waits for a message to be posted to the FLEA_INPUT_MB mailbox.
- Sets the omsg and imsg pointers to the buffers just processed.
- Verifies that the INBUF packet version and software level are compatible.
- If the dr11w_init_out debug switch is enabled, calls host_if_debug to display messages contents (if the debug display feature is enabled).

If Flea is running in FLEA_NORMAL mode, this is the only routine called to exchange packets. If a physical interface was specified, flea_host_if calls the appropriate exchange_data routine also.

Called By: _set_up_for_next_frame (through *exchange_data_sim)
 cig_config (through *exchange_data)
 db_mcc_setup (through *exchange_data)
 file_control (through *exchange_data)
 get_msg_2d (through *exchange_data)
 hw_test (through *exchange_data)
 upstart (through *exchange_data)

Routines Called: host_if_debug

```
loc_ter_msg
printf
rt_pend
sc_post
SYSERR
```

Parameters: INT_4 state

Returns: none

2.10.3.3 flea_host_if

The `flea_host_if` function establishes the communications interface for use while the system is running under Flea control, and calls the appropriate `exchange_data` routine to exchange packets each frame during the Flea exercise. This function is called at startup if the operator specified Flea mode on the command line, and the Flea mode is anything *except* `FLEA_NORMAL`.

The function call is `flea_host_if()`. The function does the following:

- Sets the *exchange function pointer to the appropriate exchange_data routine, based on the flea_mode_G variable (set by init_flea_interface). This function pointer is used only by flea_host_if.

FLEA_DR11	exchange_dr11_data
FLEA_MPV	exchange_dr11_data
FLEA_ENET	exchange_enet_data
FLEA_SCSI	exchange_scsi_data
- Prompts the user to initialize Flea via Gossip.
- Waits until the go_fly_flag variable is set (indicating that Flea is initialized).
- Posts a message to the FLEA_OUTPUT_MB mailbox.
- Waits for a message to be posted to the FLEA_INPUT_MB mailbox.
- During the Flea exercise:
 - Uses the *exchange function pointer to call the appropriate exchange_data routine each frame; the state is set to C_SIMULATION.
 - Uses the FLEA_OUTPUT_MB and FLEA_INPUT_MB mailboxes to post and receive messages.

If an invalid Flea mode is detected, the function outputs an error and exits with a 1.

Called By: main (in rtt.c)

```
Routines Called:      *exchange
                      exit
                      printf
                      rt_pend
                      sc_delay
                      sc_post
```

Parameters: none

Returns: none

2.10.3.4 init_flea_interface

The `init_flea_interface` function sets the Flea interface as the current Host interface by setting both the `*exchange_data` and `*exchange_data_sim` function pointers to `exchange_flea_data`. This function is called if Flea mode was specified at startup.

The function call is `init_flea_interface(mode)`, where *mode* is one of the following:

- 1 (FLEA_NORMAL) - no interface
- 2 (FLEA_DR0) - DR11-W interface
- 3 (FLEA_ENET) - Ethernet interface
- 4 (FLEA_MPV) - MPV interface; DR11-W interface is actually used
- 5 (FLEA_SCSI) - SCSI interface

The mode is specified by the operator on the startup command line.

`init_flea_interface` does the following:

- Sets `flea_mode_G` to the specified *mode*.
- Calls `open_flea_interface` to open the specified interface (Ethernet, DR11, etc.).
- Sets the `*exchange_data` and `*exchange_data_sim` function pointers to `exchange_flea_data`.

Called By: initialize

Routines Called: `open_flea_interface`
`printf`

Parameters: INT_4 mode

Returns: none

2.10.4 host_if_debug.c

The functions in the `host_if_debug.c` CSU are used to display SIM-to-CIG and CIG-to-SIM messages each frame. Debug message display is controlled as follows:

- Message display must be enabled through Gossip using the "display config. messages" or "display DR11W messages" option. Either option sets the `dr11w_init_out` debug switch to TRUE. This switch is initialized to FALSE and is also set to FALSE when a simulation is started.

- By default, all message types are enabled for display. (Gossip sets this on startup.) This can be changed through Gossip using the "select host i/f msgs menu" option. The user can enable all or selected message types, disable all or selected message types, or review the message types currently enabled for debug display.

If debug message display is enabled, the `cigsimio_obj` functions (in the Real-Time Processing CSC) copy all messages to a temporary buffer each frame. The `host_if_debug` functions get the messages to be displayed from this buffer. (If desired, the messages in this buffer can also be written to a disk file. This feature, called recording, is set up using the "record i/f messages menu" option on the Gossip main menu. It is implemented by the `cigsimio_obj` functions. The recorded file can be played back through Flea.)

If debug display is enabled, the contents of all selected message types are sent to stdout every frame. The actual print process is handled by the `print_msg_*` functions in the Message Processing CSC. These functions are tailored to each message type to provide a formatted, readable display.

The functions in this CSU are:

- `msg_shell_sort`
- `host_enable_all_debug_msgs`
- `host_disable_all_debug_msgs`
- `host_if_debug_init`
- `clear_line`
- `host_if_display_enabled_msgs`
- `host_list_msgs`
- `host_if_enable_debug_msgs`
- `host_if_disable_debug_msgs`
- `host_if_debug_main_menu`
- `host_if_debug_menu`
- `host_if_debug_tick`
- `host_if_debug`

2.10.4.1 `msg_shell_sort`

The `msg_shell_sort` function sorts the array of message types alphabetically by name instead of numerically by message code. The alphabetized array is `sorted_msg_ids[]`. This array is used to display the message types in alphabetical order for the user to review or change which messages are enabled for debug display.

This function is called when debug display is initialized. Alphabetical order is therefore set as the default for the user's display. The user can select an option on the Host Interface Debug menu to use numerical order instead.

The function call is `msg_shell_sort()`.

Called By: `host_if_debug_init`

Routines Called: `strcmp`

Parameters: none

Returns: none

2.10.4.2 **host_enable_all_debug_msgs**

The `host_enable_all_debug_msgs` function sets the `message_enabled` flag to TRUE for every message type. This causes all messages to be displayed each frame if debug message display is enabled. This function is called by gossip at startup, which has the effect of defaulting to all messages being enabled for display. This function is also called if the user selects the a ("all messages enabled") option from the Host Interface Debug menu.

The function call is `host_enable_all_debug_msgs()`.

Called By: gossip
host_if_debug_menu

Routines Called: none

Parameters: none

Returns: none

2.10.4.3 **host_disable_all_debug_msgs**

The `host_disable_all_debug_msgs` function sets the `message_enabled` flag to FALSE for every message type. This stops all messages from being displayed. This function is called if the user selects the z ("zero messages enabled") option from the Host Interface Debug menu.

The function call is `host_disable_all_debug_msgs()`.

Called By: host_if_debug_menu

Routines Called: none

Parameters: none

Returns: none

2.10.4.4 host_if_debug_init

The `host_if_debug_init` function initializes the array used to display the contents of all or selected message types every frame. This function is called when gossip starts up.

The function call is `host_if_debug_init()`. The function does the following:

- If `msg_debug_init_flag` is FALSE (indicating that message display has not been initialized):
 - Calls `init_print_msg_array` to initialize the `print_msg` array.
 - Sets a pointer (`message_enabled`) to the array returned by `init_print_msg_array`. (This array holds each message type's enabled flags and the name of the `print_msg` function used to print it. It is used by `host_if_debug_tick` to determine whether a given message should be displayed, and to call the correct `print_msg` function to do so.)
 - Allocates memory for a sorted list of the message ids.
 - Sets `msg_debug_init_flag` to TRUE.
- Calls `msg_shell_sort` to sort the messages alphabetically by type instead of numerically by code, for use when the list is displayed to the Gossip user. (The sort order can be toggled using an option on the Host Interface Debug menu.)
- Sets `alphaorder` to TRUE (messages are sorted alphabetically).

Called By: gossip

Routines Called: `init_print_msg_array`
`malloc`
`msg_shell_sort`

Parameters: none

Returns: none

2.10.4.5 clear_line

The `clear_line` function outputs a blank line. This function is called to clear the prompt line at the bottom of the screen before displaying a prompt to the user.

The function call is `clear_line()`.

Called By: `host_if_debug_menu`
`host_if_display_enabled_msgs`
`host_if_disable_debug_msgs`
`host_if_enable_debug_msgs`

Routines Called: `printf`

Parameters: none

Returns: none

2.10.4.6 host_if_display_enabled_msgs

The `host_if_display_enabled_msgs` function displays a list of all message types currently enabled for debug display. This function is called if the user selects the `d` ("display which messages have been enabled") option from the Host Interface Debug menu.

The function call is `host_if_display_enabled_msgs()`. The function does the following:

- Clears the screen.
- Displays the name and code of each message type currently enabled for debug display (i.e., every message type that has its `message_enabled` flag set to `TRUE`).
- Displays a message that the listed messages have been enabled for debug printing.
- Gets and discards the keystroke entered by the user (indicating that the user is ready to proceed).
- If the `dr1lw_init_out` debug switch is enabled, calls `host_if_debug` to display messages contents (if the debug display feature is enabled).

Called By: `host_if_debug_menu`

Routines Called: `blank`
`clear_line`
`cup`
`host_if_debug_main_menu`
`printf`
`unbf_getchar`

Parameters: none

Returns: none

2.10.4.7 host_list_msgs

The `host_list_msgs` function displays a list of all message types and their index numbers. The user refers to this list to select the message types to be enabled for debug display. This function is called if the user enters `?` (help) when specifying the message types to be enabled.

The function call is `host_list_msgs()`. The function does the following:

- Clears the screen.

- Displays the name and index number of each message type.

Called By: host_if_disable_debug_msgs
 host_if_enable_debug_msgs

Routines Called: blank
 cup
 printf

Parameters: none

Returns: none

2.10.4.8 host_if_enable_debug_msgs

The host_if_enable_debug_msgs function lets the user select the message types to be enabled for debug display. This function is called if the user selects the e ("enable messages to be displayed") option from the Host Interface Debug menu.

The function call is host_if_enable_debug_msgs(). The function does the following:

- Calls host_list_msgs to display all message types and their index numbers.
- Prompts the user for the index number of the message type to be enabled.
- If the user enters a valid number:
 - Sets the message_enabled flag for the specified message type to TRUE.
 - Displays a confirmation message.
- If the user enters an invalid number:
 - Displays an error message.
- If the user enters ?:
 - Calls host_list_msgs to redisplay the list.
- If the user enters x (exit):
 - Calls host_if_debug_main_menu to redisplay the Host Interface Debug menu.

Called By: host_if_debug_menu

Routines Called: atoi
 clear_line
 cup
 host_if_debug_main_menu
 host_list_msgs
 printf
 scanf
 strcpy

Parameters: none

Returns: none

2.10.4.9 host_if_disable_debug_msgs

The `host_if_disable_debug_msgs` function lets the user disable debug display for selected message types. This function is called if the user selects the `u` ("disable display of message") option from the Host Interface Debug menu.

The function call is `host_if_disable_debug_msgs()`. The function does the following:

- Calls `host_list_msgs` to display all message types and their index numbers.
- Prompts the user for the index number of the message type to be disabled.
- If the user enters a valid number:
 - If the specified message type is currently enabled, sets its `message_enabled` flag to `FALSE` and displays a confirmation message.
 - If the specified message type is already disabled, displays an error message.
- If the user enters an invalid number:
 - Displays an error message.
- If the user enters `?`:
 - Calls `host_list_msgs` to redisplay the list.
- If the user enters `x` (exit):
 - Calls `host_if_debug_main_menu` to redisplay the Host Interface Debug menu.

Called By: `host_if_debug_menu`

Routines Called: `atoi`
`clear_line`
`cup`
`host_if_debug_main_menu`
`host_list_msgs`
`printf`
`scanf`
`strcpy`

Parameters: none

Returns: none

2.10.4.10 host_if_debug_main_menu

The `host_if_debug_main_menu` function displays the Host Interface Debug menu. This function is called if the user enters `?` at the "Host Debug>" prompt. It is also called when the user finishes using an option selected from the Host Interface Debug menu (e.g., finishes selecting messages to be displayed).

The function call is **host_if_debug_main_menu()**. The function clears the screen and then displays the menu.

The option selected by the user is processed by the **host_if_debug_menu** function. Refer to **host_if_debug_menu** for a list of the options displayed on the menu.

Called By: **host_if_debug_menu**
 host_if_display_enabled_msgs
 host_if_disable_debug_msgs
 host_if_enable_debug_msgs

Routines Called: **blank**
 cup
 printf

Parameters: **none**

Returns: **none**

2.10.4.11 host_if_debug_menu

The **host_if_debug_menu** function processes the user's selection on the Host Interface Debug menu. This function is called if the user selects the **h** ("select host i/f msgs menu") option from the Gossip main menu.

The function call is **host_if_debug_menu()**. The function displays the "Host Debug>" prompt, gets the user's keystroke, and processes the user's selection (usually by calling another function).

The following table lists (in alphabetical order) the options supported by **host_if_debug_menu**, and shows the steps it performs to process each one. The menu itself is displayed by **host_if_debug_main_menu**. Options flagged with an asterisk are supported but do not appear on the menu.

Host Interface Debug Menu Option	Processing by host_if_debug_menu
?* Display this menu	Calls host_if_debug_main_menu.
a all messages enabled	Calls host_enable_all_debug_msgs.
D Display which msgs have been enabled for realtime	(not currently implemented)
d display which messages have been enabled	Calls host_if_display_enabled_msgs.
E Enable msgs to display in realtime	(not currently implemented)
e enable messages to be displayed	Calls host_if_enable_debug_msgs.
o order in which to display; alphabetic/numeric toggle	Toggles alphaorder variable; displays new state (alphabetical order or numerical order).
s edit which states to display	(not current implemented)
u disable display of message	Calls host_if_disable_debug_msgs.
x exit this submenu	Exits.
Z Zero realtime msgs enabled	(not currently implemented)
z zero messages enabled	Calls host_disable_all_debug_msgs.

Called By: gossip_tick

Routines Called: clear_line
cup
host_disable_all_debug_msgs
host_enable_all_debug_msgs
host_if_debug_main_menu
host_if_disable_debug_msgs
host_if_display_enabled_msgs
host_if_enable_debug_msgs
printf
unbf_getchar

Parameters: none

Returns: none

2.10.4.12 host_if_debug_tick

The host_if_debug_tick function is responsible for displaying the contents of each enabled message each frame. All messages from the previous frame are stored in a temporary buffer by cigsimio_write (in the Real-Time Processing CSC). host_if_debug_tick first checks to see if the message type is enabled for debug display, then calls the appropriate print_msg function (in the Message Processing CSC) to output the message to stdout. The print_msg function is called by indexing into the message_enabled[] array.

The function call is **host_if_debug_tick(pkt_P, pkt_len)**, where:

pkt_P is a pointer to the cigsimio buffer
pkt_len is the length in bytes of the cigsimio buffer

host_if_debug_tick does the following:

- Examines the tag record attached to the buffer.
 - If the tag is valid, outputs the tag (SIM=>CIG or CIG=>SIM), frame number, time, and size in bytes.
 - If the tag is invalid, outputs the tag data and exits.
- Processes each message in the packet in turn.
 - If the message type is M_LT_PIECE and MSG_LT_PIECE messages are enabled for debug display, calls print_msg_lt_piece to output the message to stdout.
 - If the message type is any other valid message and that type is enabled for debug display, indexes into the message_enabled[] array to call the appropriate print_msg function to output the message to stdout.
 - If the message type is invalid, outputs an error.

The message_enabled[] array is indexed by message type. Each array element includes the name of the print_msg function that prints that message type. For more information on this structure, see print_msg.c in the Message Processing CSC.

Called By:	debug_initdr host_if_debug	
Routines Called:	*(message_enabled[msg_type].print_msg) fflush GLOB print_msg_lt_piece printf	
Parameters:	INT_4 INT_4	*pkt_P pkt_len
Returns:	none	

2.10.4.13 host_if_debug

The host_if_debug function gets a pointer to the current frame's temporary message buffer created by cigsimio_write, then calls host_if_debug_tick to display the contents of all enabled messages. This function is called every frame by the exchange_data routines if the drllw_init_out debug switch is on. (The exchange_data routines are used to exchange message packets during every CIG state *except* simulation.) The drllw_init_out debug switch can be toggled by selecting the i ("display config. messages") option from the Gossip main menu.

host_if_debug is also called if the user selects the d ("display DR11W messages") option from any of several Gossip menus.

The function call is **host_if_debug()**. The function does the following:

- Clears any stray messages from the `cigsimio_complete_mbx` mailbox.
- Calls `cigsimio_buffer_init` to initialize the message buffer and set the `cigsimio_log_request` variable to TRUE. (This triggers the `cigsimio_obj` functions to write the messages to the temporary buffer each frame.)
- If `single_step` mode is set, calls `sysrup_on` to turn interrupts on by posting a message to the `SIMULATION_MB` mailbox.
- Waits for a message to be posted to the `cigsimio_complete_mbx` mailbox. This message is posted at the end of each frame by `cigsimio_frame_end`. It indicates that all messages from the current frame have been written to the buffer.
- Calls `cigsimio_get_data` to get a pointer to the buffer.
- Calls `host_if_debug_tick` to process the messages in the buffer.

Called By: `exchange_dr11_data`
 `exchange_enet_data`
 `exchange_flea_data`
 `exchange_mpv_data`
 `exchange_scsi_data`
 `exchange_socket_data`
 `gos_120tx`
 `gos_mpvio`
 `gos_system`
 `gossip_tick`

Routines Called: `cigsimio_buffer_init`
 `cigsimio_get_data`
 `host_if_debug_tick`
 `rt_pend`
 `sc_accept`
 `sysrup_on`

Parameters: none

Returns: none

2.10.5 `host_mpv_if.c`

The functions in the `host_mpv_if.c` CSU exchange data packets over an MPV interface. These functions are:

- `open_mpv_interface`
- `exchange_mpv_data`
- `exchange_mpv_data_sim`
- `init_mpv_interface`

MPV mode is specified by including the "m" argument on the startup command line. The operator also specifies the CIG and Simulation Host identifiers.

2.10.5.1 open_mpv_interface

The `open_mpv_interface` function establishes the MPV communication channel with the Simulation Host.

The function call is `open_mpv_interface()`. The function does the following:

- Calls `cif_init` to initialize the CIG host id.
- Calls `cif_connect` to establish the connection with the Simulation Host host id.
- Puts a dummy message into INBUF.

Called By: `init_mpv_interface`

Routines Called: `cif_connect`
`cif_init`

Parameters: `none`

Returns: `none`

2.10.5.2 exchange_mpv_data

The `exchange_mpv_data` function exchanges output and input buffers with the Simulation Host during any CIG state other than simulation. This function is called via the `*exchange_data` function pointer if MPV mode has been established.

The function call is `exchange_mpv_data(state)`, where *state* is the current state of the CIG. `exchange_mpv_data` does the following:

- If the `dr11_pkt_debug` switch is enabled, outputs the packet sizes to stdout.
- Generates a `MSG_SYS_ERROR` message to report errors in the previous frame.
- Adds a `MSG_END` message to the outgoing packet.
- Sets the outgoing packet size.
- Calls `cif_send` to write the packet from OUTBUF to the SIM host id.
- Calls `cif_receive` to read the new incoming packet from the SIM host id to INBUF.
- Resets the INBUF and OUTBUF pointers.
- If the `dr11w_init_out` debug switch is enabled, calls `host_if_debug` to display messages contents (if the debug display feature is enabled).

Called By:	<code>cig_config</code>	(through <code>*exchange_data</code>)
	<code>db_mcc_setup</code>	(through <code>*exchange_data</code>)
	<code>file_control</code>	(through <code>*exchange_data</code>)
	<code>get_msg_2d</code>	(through <code>*exchange_data</code>)
	<code>hw_test</code>	(through <code>*exchange_data</code>)

upstart (through *exchange_data)

Routines Called: cif_receive
cif_send
host_if_debug
printf

Parameters: INT_4 state

Returns: none

2.10.5.3 exchange_mpv_data_sim

The `exchange_mpv_data_sim` function exchanges output and input buffers with the Simulation Host during a simulation. This function is called (via the `*exchange_data_sim` function pointer) at the end of every frame if MPV mode has been established.

The function call is `exchange_mpv_data_sim(state)`, where *state* is the current state of the CIG. `exchange_mpv_data_sim` does the following:

- If the `dr11_pkt_debug` switch is enabled, outputs the packet sizes to stdout.
- Calls `loc_ter_msg` to generate a local terrain message if one is required this frame.
- Adds a `MSG_END` message to the outgoing packet.
- Sets the outgoing packet size.
- Calls `cif_send` to write the packet from OUTBUF to the SIM host id.
- Calls `cif_receive` to read the new incoming packet from the SIM host id to INBUF.
- Resets the INBUF and OUTBUF pointers.

Called By: _set_up_for_next_frame (through *exchange_data_sim)

Routines Called: cif_receive
cif_send
loc_ter_msg
printf

Parameters: INT_4 state

Returns: none

2.10.5.4 init_mpv_interface

The `init_mpv_interface` function establishes MPV as the current interface to the Simulation Host. This function is called if MPV mode was specified at startup.

The function call is `init_mpv_interface(mine, his)`, where:

mine is the CIG's host id specified at startup
his is the Simulation Host's host id specified at startup

init_mpv_interface does the following:

- Sets the host ids: cig_host_if_id (CIG) and sim_host_if_id (Simulation Host).
- Calls open_mpv_interface to open the communications path.
- Calls host_if_buffer_init to initialize the output buffer header (without reinitializing the buffers).
- Sets the *exchange_data function pointer to exchange_mpv_data.
- Sets the *exchange_data_sim function pointer to exchange_mpv_data_sim.

Called By: initialize

Routines Called: host_if_buffer_init
 open_mpv_interface
 printf

Parameters: INT_4 mine
 INT_4 his

Returns: none

2.10.6 host_scsi_if.c

The functions in the host_scsi_if.c CSU exchange data packets over a SCSI physical interface. These functions are:

- open_scsi_interface
- exchange_scsi_data
- exchange_scsi_data_sim
- init_scsi_interface

SCSI interface mode is set by including the "z" argument on the startup command line.

2.10.6.1 open_scsi_interface

The open_scsi_interface function opens the initiator channel for writing data out to the Simulation Host, and the target channel for receiving data from the Simulation Host.

The function call is open_scsi_interface(). The function does the following:

- Opens the SCSI device as the init_fd (initiator channel) in write-only mode.
- Opens the CIG as the targ_fd (target channel) in read-only mode.

If either device cannot be opened, the function outputs an error and exits with a 1.

Called By: init_scsi_interface
 open_flea_interface

Routines Called: exit
 open
 printf
 sprintf

Parameters: none

Returns: none

2.10.6.2 exchange_scsi_data

The exchange_scsi_data function exchanges output and input buffers with the Simulation Host during any CIG state other than simulation. This function is called via the *exchange_data function pointer if SCSI mode has been established. It is also called via the *exchange function pointer if the system is running under Flea and SCSI mode was selected.

The function call is **exchange_scsi_data(state)**, where *state* is the current state of the CIG. exchange_scsi_data does the following:

- If the scsi_pkt_debug switch is enabled, outputs the packet sizes to stdout.
- Calls SYSERR to generate a MSG_SYS_ERROR message to report errors in the previous frame.
- Adds a MSG_END message to the outgoing packet.
- Sets the outgoing packet size.
- Writes the outgoing packet from OUTBUF to init_fd.
- Reads the new incoming packet from targ_fd into INBUF.
- Resets the INBUF and OUTBUF pointers.
- If the dr1lw_init_out debug switch is enabled, calls host_if_debug to display messages contents (if the debug display feature is enabled).

Called By:	cig_config	(through *exchange_data)
	db_mcc_setup	(through *exchange_data)
	file_control	(through *exchange_data)
	flea_host_if	(through *exchange)
	get_msg_2d	(through *exchange_data)
	hw_test	(through *exchange_data)
	upstart	(through *exchange_data)

Routines Called: host_if_debug
 printf
 read
 SYSERR
 write

Parameters: . INT_4 state

Returns: none

2.10.6.3 exchange scsi data sim

The `exchange_scsi_data_sim` function exchanges output and input buffers with the Simulation Host during a simulation. This function is called (via the `*exchange_data_sim` function pointer) at the end of every frame if SCSI mode has been established.

The function call is `exchange_scsi_data_sim(state)`, where *state* is the current state of the CIG. `exchange_scsi_data_sim` does the following:

- If the `scsi_pkt_debug` switch is enabled, outputs the packet sizes to stdout.
- Calls `loc_ter_msg` to generate a local terrain message if one is required this frame.
- Calls `SYSERR` to generate a `MSG_SYS_ERROR` message to report errors in the previous frame.
- Adds a `MSG_END` message to the outgoing packet.
- Sets the outgoing packet size.
- Writes the outgoing packet from `OUTBUF` to `init_fd`.
- Reads the new incoming packet from `targ_fd` into `INBUF`.
- Resets the `INBUF` and `OUTBUF` pointers.

Called By: set up for next frame (through *exchange_data_sim)

```
Routines Called:  loc_ter_msg
                  printf
                  read
                  SYSERR
                  write
```

Parameters: INT 4 state

Returns: none

2.10.6.4 init scsi interface

The `init_scsi_interface` function establishes SCSI as the current interface. This function is called at startup if SCSI mode was specified on the command line.

The function call is `init_scsi_interface()`. The function does the following:

- Calls `open_scsi_interface` to open the SCSI device channels.
- Sets the `*exchange_data` function pointer to `exchange_scsi_data`.
- Sets the `*exchange_data_sim` function pointer to `exchange_scsi_data_sim`.

Called By: initialize

Routines Called: open_scsi_interface
printf

Parameters: none

Returns: none

2.10.7 host_socket_if.c

The functions in the host_socket_if.c CSU exchange data packets over a Socket physical interface. These functions are:

- open_socket_interface
- exchange_socket_data
- exchange_socket_data_sim
- init_socket_interface

Socket mode is specified with the "x" argument on the startup command line. The user also enters the socket interface name.

2.10.7.1 open_socket_interface

The open_socket_interface function opens the socket interface as host_sd.

The function call is **open_socket_interface()**. The function does the following:

- Calls ssocket to open the local_db socket as host_sd.
- Calls sibuffer to set the input and output buffer sizes.
- Calls sbind to bind the socket name to the host_sd file.
- Calls slisten and saccept to get an acknowledgement from the host_sd file.
- Initializes the OUTBUF header.
- Resets the OUTBUF pointer.

If the socket cannot be opened, the function outputs an error and exits with a -1.

Called By: init_socket_interface

Routines Called: exit
perror
saccept
sbind
sgbuffer
sibuffer

slisten
ssocket

Parameters: none

Returns: none

2.10.7.2 exchange_socket_data

The `exchange_socket_data` function exchanges output and input buffers with the Simulation Host during any CIG state other than simulation. This function is called via the `*exchange_data` function pointer if socket mode has been established.

The function call is `exchange_socket_data(state)`, where *state* is the current state of the CIG. `exchange_socket_data` does the following:

- If the `dr11_pkt_debug` switch is enabled, outputs the packet sizes to stdout.
- Generates a `MSG_SYS_ERROR` message to report errors in the previous frame.
- Adds a `MSG_END` message to the outgoing packet.
- Sets the outgoing packet size.
- Sends `OUTBUF` to `host_sd`.
- Calls `sgbuffer` to get the buffer from the `host_sd`.
- Initializes the `OUTBUF` packet header.
- Calls `spbuffer` to put the buffer in the `host_sd`.
- Calls `srecv` to receive the new incoming buffer from `host_sd`.
- Puts the buffer in `INBUF`.
- Resets the `INBUF` and `OUTBUF` pointers.
- If the `dr11w_init_out` debug switch is enabled, calls `host_if_debug` (to display messages contents if the debug display feature is enabled).

Called By:	<code>cig_config</code>	(through <code>*exchange_data</code>)
	<code>db_mcc_setup</code>	(through <code>*exchange_data</code>)
	<code>file_control</code>	(through <code>*exchange_data</code>)
	<code>get_msg_2d</code>	(through <code>*exchange_data</code>)
	<code>hw_test</code>	(through <code>*exchange_data</code>)
	<code>upstart</code>	(through <code>*exchange_data</code>)

Routines Called:

`host_if_debug`
`printf`
`sgbuffer`
`spbuffer`
`srecv`
`ssend`

Parameters: `INT_4` `state`

Returns: none

2.10.7.3 exchange_socket_data_sim

The `exchange_socket_data_sim` function exchanges output and input buffers with the Simulation Host during a simulation. This function is called (via the `*exchange_data_sim` function pointer) at the end of every frame if Socket mode has been established.

The function call is `exchange_socket_data_sim(state)`, where *state* is the current state of the CIG. `exchange_socket_data_sim` does the following:

- If the `dr11_pkt_debug` switch is enabled, outputs the packet sizes to `stdout`.
- Calls `loc_ter_msg` to generate a local terrain message if one is required this frame.
- Generates a `MSG_SYS_ERROR` message to report errors in the previous frame.
- Adds a `MSG_END` message to the outgoing packet.
- Sets the outgoing packet size.
- Sends `OUTBUF` to `host_sd`.
- Calls `sgbuffer` to get the buffer from the `host_sd`.
- Initializes the `OUTBUF` packet header.
- Calls `spbuffer` to put the buffer in the `host_sd`.
- Calls `srecv` to receive the new incoming buffer from `host_sd`.
- Puts the buffer in `INBUF`.
- Resets the `INBUF` and `OUTBUF` pointers.

Called By: `_set_up_for_next_frame` (through `*exchange_data_sim`)

Routines Called: `loc_ter_msg`
`printf`
`sgbuffer`
`spbuffer`
`srecv`
`ssend`

Parameters: `INT_4` `state`

Returns: `none`

2.10.7.4 init_socket_interface

The `init_socket_interface` function establishes socket mode as the current interface. This function is called if a socket interface name was specified at startup. It is also called if AGPT mode was selected. (AGPT mode is a non-standard mode that is not addressed in this document.)

The function call is `init_socket_interface()`. The function does the following:

- Calls `open_socket_interface` to open the socket.
- Sets the `*exchange_data` function pointer to `exchange_socket_data`.
- Sets the `*exchange_data_sim` function pointer to `exchange_socket_data_sim`.

Called By: agpt_init
 initialize

Routines Called: open_socket_interface
 printf

Parameters: none

Returns: none

2.11 MPV Interface (/cig/libsrc/libmpvideo)

The MPV Interface CSC is responsible for initializing and communicating with the Micro Processor Video (MPV) board. This board is present in TX backends only; therefore, the functions in this CSC pertain to TX backends only.

The MPV is the final board in the TX graphics pipeline. It contains the GSP (Graphics System Processor) chip, which handles 2-D overlays. The MPV board is responsible for the following:

- Calculating pixel depth for laser range requests.
- Loading color lookup tables as required.
- Setting video control registers.
- Reading from and writing to GSP memory.
- Starting and stopping the GSP.

The following files are downloaded to the GSP:

task2d	The 2-D processor task for 2-D overlays.
data2d	The 2-D overlay database.
lookut	Color lookup table.
bitmap	<<TBD>>

The MPV Interface routines do not communicate directly with either the MPV board or the GSP. All communication is handled by the Force Processing task running on the Force board. The MPV Interface functions send messages and commands to the Force task, and the Force task downloads the requests to the MPV board or GSP memory. Responses from the MPV and the GSP are also routed back through the Force board. (The Force Processing CSC is described in section 2.18.)

The MPV Interface routines and the Force board communicate using two different interface methods: message buffers and the Force-MPV mailbox.

Message Buffers

Some messages are passed using half-word message buffers. Messages sent from the real-time software (via the MPV Interface routines) to Force are prefixed with MSG_F0. Messages returned from Force are prefixed with MSG_F1.

Both the MPV Interface functions and the Force functions use the routines in the mx2_hword.c CSU to manage and use the message buffers. These routines are part of both the Real-Time Processing CSC and the Force Processing CSC.

The message buffers used by the MPV Interface routines to communicate with the Force task are the following:

mpvio_to_buf (outgoing)

Used for messages sent from the MPV Interface routines to Force. These messages result from changes requested by the Simulation Host to modify simulation parameters.

mpvio_from_buf (incoming)

Used for response messages returned by Force to the MPV Interface routines.

Force-MPV Mailbox

For some functions, the MPV Interface routines place commands directly into the Force board's front-end control register (FE_CONTROL). This interface uses an intertask mailbox. The commands are queued in the Force-MPV mailbox until the end of the frame, at which time a message is sent to Force to trigger processing of the queued commands.

The structure of the Force-MPV mailbox (MPVIO_INTERFACE) is defined in the mpvideo.h file. The mailbox includes locations for the Force front-end control register, the force control register, laser range pixel locations and returned depth values, GSP addresses (for read and write requests), lookup table specifications (wanted table, loaded table, and table addresses), various timing parameters, and the buffers used to exchange PASS_ON and PAS_BACK data.

Messages passed between the real-time software and the 2-D processor task (on the GSP) are handled as follows:

- The Simulation Host sends "pass on" messages to update 2-D overlays during the simulation. The message is received by process_a_msg, which calls an MPV Interface routine. The MPV Interface function places the message in the data_buf_out buffer in the Force-MPV mailbox.
- After processing a "pass on" message, the GSP sends a "pass back" message in reply. The message is placed in the data_buf_in buffer in the Force-MPV mailbox. The MPV Interface routine sets a pointer to the message for return to the Simulation Host.
- The above process applies only to changes to the 2-D overlays. The original overlays created before the simulation starts (using the 2-D Overlay Compiler) are not passed through the MPV Interface routines. The 2-D Overlay Compiler's linkup function places the GSP download commands directly into the Force-MPV mailbox.

The Gossip user can interact with the MPV and the GSP through the gos_mpvio function. gos_mpvio uses different message buffers to communicate with the Force task. The MPV Interface routines are not involved in the Gossip process.

Figure 2-15 identifies the CSUs in the MPV Interface CSC. The functions performed by these CSUs are described in this section.

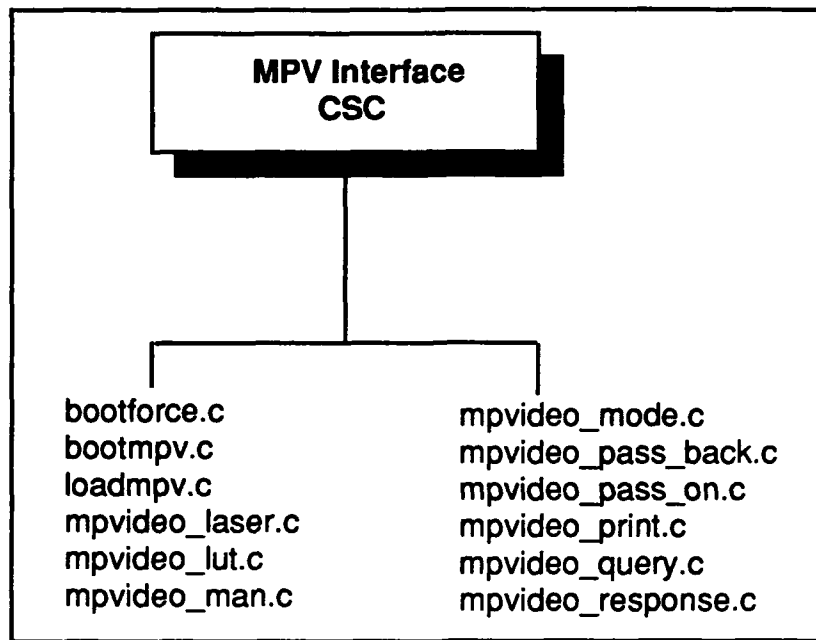


Figure 2-15. MPV Interface CSUs

2.11.1 bootforce.c

The functions in the bootforce.c CSU are used to start the Force task on the Force board, and to find the message buffers used by the MPV Interface routines to communicate with the Force task. These functions are:

- bootforce
- set_entry_pt
- rtn_entry_pt
- mpvmsg_query_buf_addr
- mpvmsg_reply_buf_addr
- mpvmsg_to_buf_addr
- mpvmsg_from_buf_addr

2.11.1.1 bootforce

The bootforce function loads the Force task onto the Force board in a specified backend, which causes the Force task to start up. It also sets up the addresses of the buffers used by the MPV Interface routines to communicate with the Force board. This function is called when the MPV is initialized.

The function call is **bootforce(backend, file_name)**, where:

backend is the backend id

file_name is the name of the executable S-record file to be loaded onto the Force board

bootforce does the following:

- Calls `mpvideo_get_object_addr` to get a pointer to the MPV object in the specified backend.
- Checks to make sure Force has not already been started.
- Clears the Force memory.
- Calls `sload` to load the specified file.
- Sets the `start_force` variable to TRUE.
- Waits for the Force task to start up.
- Sets the addresses of the message buffers used to communicate with the Force board (the incoming and outgoing buffers used by the MPV Interface routines, as well as the query and reply buffers used by `gos_mpvio`).

The function returns TRUE if successful. It returns FALSE if it cannot get a pointer to the MPV object, or if the Force file cannot be loaded.

Called By: `mpvideo_boot`

Routines Called: `mpvideo_get_object_addr`
`printf`
`sc_delay`
`sload`
`status_mpvideo_print`

Parameters: `UNS_4` `backend`
`UNS_1` `*file_name`

Returns: 1 (TRUE)
0 (FALSE)

2.11.1.2 `set_entry_pt`

The `set_entry_pt` function sets an entry point for <<TBD>>

The function call is `set_entry_pt(backend, entry_pt)`, where:

backend is the backend id
entry_pt is <<TBD>>>

`set_entry_pt` does the following:

- Calls `mpvideo_get_object_addr` to get a pointer to the MPV object in the specified backend.
- Sets the high and low entry points.

The function returns TRUE if successful. It returns FALSE if it could not get a pointer to the MPV object.

Called By: `mpvideo_load`

Routines Called: mpvideo_get_object_addr

Parameters: UNS_4 backend
 UNS_4 entry_pt

Returns: 1 (TRUE)
 0 (FALSE)

2.11.1.3 rtn_entry_pt

The `rtn_entry_pt` function returns the entry point that was set with `set_entry_pt`.

The function call is `rtn_entry_pt(backend, entry_pt_h, entry_pt_l)`, where:

backend is the backend id

entry_pt_h is the location for the returned high entry point

entry_pt_l is the location for the returned low entry point

`rtn_entry_pt` does the following:

- Calls `mpvideo_get_object_addr` to get a pointer to the MPV object in the specified backend.
- Places the high and low entry points into the locations specified in the call.

The function returns TRUE if successful. It returns FALSE if it could not get a pointer to the MPV object.

Called By: mpvideo_sim_init

Routines Called: mpvideo_get_object_addr

Parameters: UNS_4 backend
 UNS_2 *entry_pt_h
 UNS_2 *entry_pt_l

Returns: 1 (TRUE)
 0 (FALSE)

2.11.1.4 mpvmsg_query_buf_addr

The `mpvmsg_query_buf_addr` function returns the address of the Force query buffer for a specified backend. The query buffer is used by `gos_mpvio` (part of Gossip) to send messages to the Force task.

The function call is **mpvmsg_query_buf_addr(backend)**, where *backend* is the backend id.

The function returns FALSE if it cannot get a pointer to the MPV object.

Called By:	gos_mpvio	
Routines Called:	mpvideo_get_object_addr	
Parameters:	UNS_4	backend
Returns:	pmpv->force_query_addr 0 (FALSE)	

2.11.1.5 mpvmsg_reply_buf_addr

The **mpvmsg_reply_buf_addr** function returns the address of the Force reply buffer for a specified backend. The reply buffer is used for messages returned to gos_mpvio from Force.

The function call is **mpvmsg_reply_buf_addr(backend)**, where *backend* is the backend id.

The function returns FALSE if it could not get a pointer to the MPV object.

Called By:	gos_mpvio	
Routines Called:	mpvideo_get_object_addr	
Parameters:	UNS_4	backend
Returns:	pmpv->force_reply_addr 0 (FALSE)	

2.11.1.6 mpvmsg_to_buf_addr

The **mpvmsg_to_buf_addr** function returns the address of the outgoing (to Force) buffer for a specified backend. Most of the MPV Interface routines use this buffer to send messages to the Force board.

The function call is **mpvmsg_to_buf_addr(backend)**, where *backend* is the backend id.

The function returns FALSE if it could not get a pointer to the MPV object.

Called By: none

Routines Called: mpvideo_get_object_addr

Parameters: UNS_4 backend

Returns: pmpv->to_force_addr
0 (FALSE)

2.11.1.7 mpvmsg_from_buf_addr

The mpvmsg_from_buf_addr function returns the address of the incoming (from Force) buffer for a specified backend. The Force board uses this buffer to send replies back to the MPV Interface routines.

The function call is mpvmsg_from_buf_addr(backend), where *backend* is the backend id.

The function returns FALSE if it could not get a pointer to the MPV object.

Called By: gos_mpvio

Routines Called: mpvideo_get_object_addr

Parameters: UNS_4 backend

Returns: pmpv->from_force_addr
0 (FALSE)

2.11.2 bootmpv.c

The functions in the bootmpv.c CSU are responsible for initializing the MPV board. These functions are:

- mpvideo_boot
- prtmsgerr
- prtackerr
- prtstaterr

This CSU also defines the WAIT_MPVREPLY macro, described in Appendix B.

2.11.2.1 mpvideo_boot

The `mpvideo_boot` function sets up the MPV I/O board and determines MPV board status. This function is called when the system is initialized if a `force0` file is found on disk.

The function call is `mpvideo_boot (backend, boot_io)`, where:

backend is the backend id

boot_io is the name of the S-record executable file to be loaded onto the MPV I/O board

`mpvideo_boot` does the following:

- Calls `mpvideo_get_object_addr` to get a pointer to the MPV object in the specified backend.
- Calls `bus_error` to verify that the board's base address exists.
- Calls `bootforce` to load the MPV I/O board with the specified file.
- Sets a pointer to the MPV's mailbox.
- Locates the message buffers used to communicate with the Force task.
- Makes sure the message interface buffers appear to be valid.
- Calls `mx2_peek` to preview the top message in the incoming buffer, then deletes it with `mx2_skip`.
- Pushes a `MSG_F0_MPV_LUT_TYPE_REQUEST` message onto the outgoing message buffer; this message asks for the MPV board revision level and whether or not the 2-D (final) lookup tables are downloadable.
- Uses the `WAIT_MPVREPLY` macro to wait for a `MSG_F1_MPV_LUT_TYPE` response message from Force.
- When the reply is received, sets the MPV type and lut download flag.
- If the MPV board is a Value-Added MPV:
 - Pushes a `MSG_F0_MPV_RESET` message onto the outgoing buffer; this message resets the MPV board.
 - Waits for an acknowledgement from Force.
- Halts the GSP task (even if it is not running) by writing a `SUBSYS_STOP` command into the Force task's front-end control register.
- Waits for a response from Force.
- Runs a test on GSP memory by writing a `SUBSYS_TEST_MEM` command into the Force task's front-end control register.
- Waits for a response from Force.
- Sets the control register in the MPV by writing a `SUBSYS_WRITE_START` command into the Force task's front-end control register.
- Waits for a response from Force.
- If the MPV board is a Value-Added MPV:
 - Calls `mpvideo_define_mode` to define the default resolution modes.
 - Calls `mpvideo_set_mode` to put one of the default resolution modes into effect.
 - Waits for an acknowledgement from Force.
 - Sets the display configuration to 0.
- If the MPV board is an older version:
 - Uses the `SUBSYS_READ_START` command to read GSP memory (via the Force control register) to determine if the backend is displaying high resolution or low resolution.
 - Waits for a response from Force.
 - Sets the display configuration appropriately.

The function returns 0 if successful. It returns EOF if it could not get a pointer to the MPV, the board's base address was not found, the boot file could not be loaded, the GSP memory test failed, or the MPV I/O control write failed.

Called By:	initialize	
Routines Called:	bootforce bus_error mpvideo_define_mode mpvideo_get_object_addr mpvideo_set_mode mx2_peek mx2_push mx2_skip printf prtackerr prtmsgerr sc_delay status_mpvideo_print WAIT_MPVIO WAIT_MPVREPLY	
Parameters:	UNS_4 UNS_1	backend *boot_io
Returns:	EOF 0	

2.11.2.2 prtmsgerr

The prtmsgerr function outputs an "UNKNOWN MESSAGE" error for mpvideo_boot. This function is called if mpvideo_boot receives an unexpected message code from the Force task.

The function call is **prtmsgerr(msg_code)**, where *msg_code* is the unknown message code.

The error message output by prtmsgerr includes the message code.

Called By:	mpvideo_boot	
Routines Called:	printf	
Parameters:	UNS_2	msg_code

Returns: none

2.11.2.3 prtackerr

The prtackerr function outputs an "ACKNOWLEDGE ERROR" for mpvideo_boot. This function is called if mpvideo_boot receives a MSG_F1_ACKNOWLEDGE message from Force, but (1) the message does not appear to be in response to the message sent by the MPV, or (2) the message's acknowledgement code indicates a problem.

The function call is **prtackerr(msg_code, msg_P)**, where:

msg_code is the message code (MSG_F1_ACKNOWLEDGE)
msg_P is a pointer to the message

The error message output by prtackerr includes the message code and the acknowledgement code from the message.

Called By: mpvideo_boot

Routines Called: printf

Parameters: UNS_2 msg_code
UNS_4 *msg_P

Returns: none

2.11.2.4 prtstaterr

The prtstaterr function outputs a "STATUS ERROR" message for mpvideo_boot. This function is called if an error is detected in a MSG_F1_STATUS message returned from Force. (This message is not currently sent by any Force function.)

The function call is **prtstaterr(msg_code, msg_P)**, where:

msg_code is the message code
msg_P is a pointer to the message

The error message output by prtstaterr includes the message code and the status code from the message.

This function is not currently used.

Called By: none

Routines Called: printf

Parameters: UNS_2 msg_code
 UNS_4 *msg_P

Returns: none

2.11.3 loadmpv.c (mpvideo_load)

The mpvideo_load function downloads a specified file directly to the MPV in a specified backend. This function is called to download the 3-D and final (2-D) color lookup tables, the 2-D task file, and the 2-D data file. It is called at initialization time (to load default files or the files specified in the subsys.cfg file), during the database setup state (in response to MSG_FILE_DESCR messages received from the Simulation Host), and by various options available through Gossip.

The function call is mpvideo_load(backend, mpv_file), where:

backend is the backend id

mpv_file is the name of the file to be downloaded

mpvideo_load does the following:

- Calls mpvideo_get_object_addr to get a pointer to the MPV object in the specified backend.
- Calls bus_error to verify that the MPV board's base address exists.
- Opens the specified *mpv_file*.
- Reads the file header to validate its size and determine the file type.
- If the file type is "final":
 - Finds the file size in the file header.
 - Sets up initial data that will not change.
 - Reads the file a block (512 bytes) at a time and swaps the bytes.
 - Calls mx2_peek and mx2_skip to flush all messages from the incoming buffer.
 - Sends the file to the MPV via the Force board by pushing a MSG_F0_FINAL_LUT_DOWNLOAD message onto the outgoing buffer.
 - Waits for an acknowledgement from Force.
 - Verifies the block count returned.
 - Closes the file.
- If the file type is "bitmap":
 - Finds the GSP load address in the file header.
 - Resets the data area pointer.
 - Reads the file length from the file header.
 - Reads the file a block (512 bytes) at a time and swaps the bytes.
 - Downloads the data to GSP memory.
 - Closes the file.
- If the file type is "lut":
 - Finds the GSP load address in the file header.
 - Resets the data area pointer.
 - Reads the file length from the file header.
 - Reads the file a block (512 bytes) at a time.
 - Moves the data into MPV I/O board memory space for later download.

- Closes the file.
- If the file type is "2d-task":
 - Finds the GSP load address in the file header.
 - Resets the data area pointer.
 - Reads the file length from the file header.
 - Reads the file a block (512 bytes) at a time and swaps the bytes.
 - Downloads the data to GSP memory.
 - Closes the file.
 - Finds the entry point in the file header and calls set_entry_pt to set it.
- If the file type is "2d-data":
 - Finds the GSP load address in the file header.
 - Resets the data area pointer.
 - Checks the override_2d flag to see if the 2-D database has already been generated; does not load this file if the flag is set. (compile_2d in the 2-D Overlay Compiler CSC sets override_2d to TRUE after the 2-D database has been compiled.)
 - Reads the file length from the file header.
 - Reads the file a block (512 bytes) at a time.
 - Downloads the data to GSP memory.
 - Closes the file.

The function returns TRUE if successful. It returns EOF if it cannot get a pointer to the MPV, the board's base address cannot be found, or the specified file cannot be opened. It returns FALSE if the file header is missing a required data item or contains invalid data.

Called By: db_mcc_setup
 gos_120tx
 gos_mpvio
 initialize

Routines Called: bus_error
 close
 exit
 FindField
 mpvideo_get_object_addr
 mx2_hwcopu
 mx2_peek
 mx2_push
 mx2_skip
 open
 printf
 read
 sc_delay
 set_entry_pt
 sscanf
 status_mpvio_print
 strncmp
 strtol
 WAIT_MPVIO

Parameters: UNS_4

backend

UNS_1

*mpv_file

Returns: EOF
 1 (TRUE)
 0 (FALSE)

2.11.4 mpvideo_laser.c (mpvideo_laser_request_range)

The mpvideo_laser_request_range function informs the MPV that the laser range value at a specified pixel on a specified channel is to be reported every frame. This function is called when the Simulation Host sends a MSG_LASER_REQUEST_RANGE message for a TX backend.

The function call is mpvideo_laser_request_range(backend, channel, i, j, id), where:

backend is the backend id

channel is the number of the channel

i is the horizontal coordinate of the pixel for which range values are requested

j is the vertical coordinate of the pixel for which range values are requested

id is an identifier to be attached to the laser return message

mpvideo_laser_request_range does the following:

- Calls mpvideo_get_object_addr to get a pointer to the MPV object in the specified backend.
- Finds the outgoing (to Force) message buffer.
- Passes the request to Force by pushing a MSG_F0_PIXEL_DEPTH_REQUEST message onto the outgoing message buffer.
- Sets the MPV's io_req flag to TRUE if it is not already set.

The function returns 0 if successful. It returns EOF if it cannot get a pointer to the MPV object.

Called By: backend_laser_request_range

Routines Called: mpvideo_get_object_addr
 mx2_push

Parameters:	UNS_4	backend
	UNS_4	channel
	INT_2	i
	INT_2	j
	INT_2	id

Returns: 0
 EOF

2.11.5 mpvideo_lut.c (mpvideo_set_lut)

The mpvideo_set_lut function is used to change the 3-D lookup table (lut3d) and final lookup table (lut2d) indices. This function is called when the Simulation Host sends a MSG_SUBSYS_MODE message to change lookup tables.

The function call is mpvideo_set_lut(backend, channel, lut3d, lut2d), where:

backend is the backend id
channel is the channel number (0 = all channels)
lut3d is the new 3-D lookup table index
lut2d is the new final lookup table index

mpvideo_set_lut does the following:

- Calls mpvideo_get_object_addr to get a pointer to the MPV object in the specified backend.
- Validates the channel number.
- Finds the outgoing (to Force) buffer.
- Determines whether one or both lookup table values are different from their previous values.
 - If both tables are to be changed, pushes a MSG_F0_ALLLUT_SWITCH message onto the outgoing message buffer, for one or both channels.
 - If only the 2-D table is to be changed, pushes a MSG_F0_FINAL_LUT_SWITCH message onto the outgoing message buffer, for one or both channels.
 - If only the 3-D table is to be changed, pushes a MSG_F0_3DLUT_SWITCH message onto the outgoing message buffer, for one or both channels.

The function returns EOF if the channel number is invalid or the outgoing buffer cannot be found.

Called By: backend_set_color

Routines Called: mpvideo_get_object_addr
 mx2_push
 printf

Parameters:	UNS_4	backend
	UNS_2	channel
	UNS_2	lut3d
	UNS_2	lut2d

Returns: EOF

2.11.6 mpvideo_man.c

The functions in the mpvideo_man.c CSU are used to load and communicate with the MPV objects in the CIG backend. These functions are:

- mpvideo_set_video
- mpvideo_setup
- mpvideo_stop
- mpvideo_sim_init
- mpvideo_send_req
- mpvideo_get_object_addr

2.11.6.1 mpvideo_set_video

The mpvideo_set_video function turns the video channels driven by the MPV board on or off. This function is called at the beginning of a simulation (to turn all channels on) and at the end of a simulation (to turn all channels off). It is also called during a simulation if the Simulation Host sends a MSG_VIEWPORT_UPDATE message.

The function call is **mpvideo_set_video(backend, flag)**, where:

backend is the backend id

flag is 0 (off) or 1 (on)

mpvideo_set_video does the following:

- Calls mpvideo_get_object_addr to get a pointer to the MPV object in the specified backend.
- Pushes a MSG_F0_SET_DISPLAY message onto the outgoing message buffer.

Called By: backend_reset
 backend_set_video
 mpvideo_sim_init

Routines Called: mpvideo_get_object_addr
 mx2_push

Parameters: UNS_4 backend
 UNS_4 flag

Returns: none

2.11.6.2 mpvideo_setup

The `mpvideo_setup` function sets up internal control structures for the MPV board on a specified backend. This function is called as part of the backend initialization process, prior to any other MPV Interface routines.

The function call is `mpvideo_setup(backend, pboot, pmbx)`, where:

backend is the backend id

pboot is the base VMEbus address of the MPV I/O board

pmbx is the VMEbus address of the MPV I/O runtime mailbox

`mpvideo_setup` does the following:

- Calls `bus_error` to verify that the board's base address exists.
- Allocates memory for the MPV object.
- Initializes the MPV object.
- Tries to turn the video off (in case the MPV I/O board is running).

The function returns 0 if successful. It returns 1 if the MPV board could not be found.

Called By:	<code>backend_setup</code>		
Routines Called:	<code>bus_error</code> <code>malloc</code> <code>printf</code>		
Parameters:	<code>UNS_4</code> <code>UNS_2</code> <code>MPVIO_INTERFACE</code>	<code>backend</code> <code>*pboot</code> <code>*pmbx</code>	
Returns:	0 1		

2.11.6.3 mpvideo_stop

The `mpvideo_stop` function sends a stop command to the MPV board in a specified backend. This command halts execution of the MPV's 2-D processor task. This function is called when the backend is reset at the end of a simulation. It is also called if the Gossip user selects the R ("Reload MPV files & task") option from the 120TX menu.

The function call is `mpvideo_stop(backend)`, where *backend* is the backend id.

`mpvideo_stop` does the following:

- Calls `mpvideo_get_object_addr` to get a pointer to the MPV object in the specified backend.

- Puts a SUBSYS_STOP command in the Force front-end control register (via the MPV-Force interface mailbox).
- Sets the MPV's io_req flag to FALSE.
- Sets the MPV's pass_on_sent flag to FALSE.

Called By: backend_reset
gos_120tx

Routines Called: mpvideo_get_object_addr

Parameters: UNS_4 backend

Returns: `none`

2.11.6.4 mpvideo_sim_init

The `mpvideo_sim_init` function sets up internal MPV status variables, triggers the start of the 2-D processor task on the MPV board, and initializes video control registers and lookup tables. This function is called at the beginning of a simulation. It is also called if the Gossip user selects the **R** ("Reload MPV files & task") option from the 120TX menu.

The function call is **mpvideo_sim_init()**. For each MPV, mpvideo_sim_init does the following:

- Sets the MPV's `io_req` flag to `FALSE`.
- Sets the MPV's `pass_on_sent` flag to `FALSE`.
- Calls `mpvideo_get_object_addr` to get a pointer to the MPV object in the specified backend.
- Calls `rtn_entry_pt` to get the <<TBD>> entry point.
- Puts a `SUBSYS_NMI_START` command into the Force front-end control register (via the MPV-Force mailbox).
- Waits for the GSP task to start.
- Initializes the `pass_on` message length and sets the end of the message.
- Sets the video controls for each channel.
- Sets the `mpvio_to_buf` variable to the location of the outgoing (to Force) message buffer.
- Pushes a `MSG_F0_3DLUT_DOWNLOAD` message onto the outgoing buffer (to download the 3-D color lookup table).
- Calls `mpvideo_set_video` to turn the video channels on.

Called By: backend_sim_init
gos_120tx

Routines Called: mpvideo_get_object_addr
mpvideo_set_video
mx2_push
rtn_entry_pt

sc_delay
WAIT_MPVIO

Parameters: none

Returns: none

2.11.6.5 mpvideo_send_req

The mpvideo_send_req function triggers the MPV I/O to send all queued messages to the Force board. This function is called after all of a frame's messages have been processed.

The function call is mpvideo_send_req(). For each MPV, the function does the following:

- Calls mpvideo_get_object_addr to get a pointer to the MPV object in the specified backend.
- If the MPV has messages to be processed:
 - Makes sure that the Force front end is ready for a message.
 - Puts a SUBSYS_MAIL_SEND command in the MPV-Force mailbox.
 - Pushes a MSG_F0_TRIGGER message onto the outgoing message buffer.

Called By: backend_send_req

Routines Called: mpvideo_get_object_addr
mx2_push

Parameters: none

Returns: none

2.11.6.6 mpvideo_get_object_addr

The mpvideo_get_object_addr function returns a pointer to the MPV object in a specified backend. This pointer must be used for all operations affecting the MPV.

The function call is mpvideo_get_object_addr(backend), where *backend* is the backend id.

The pointers for all MPV objects in the CIG are maintained in the mpv_table[] array, which is built by mpvideo_setup. The array is indexed by backend. If the specified backend is not in the array, mpvideo_get_object_addr returns NULL.

Called By: cig_2d_setup
esifa_sim_init

```

mpvideo_boot
mpvideo_laser_request_range
mpvideo_load
mpvideo_num_paths
mpvideo_pass_back
mpvideo_pass_on
mpvideo_send_req
mpvideo_set_lut
mpvideo_set_mode
mpvideo_set_video
mpvideo_sim_init
mpvideo_stop

```

Routines Called: none

Parameters: UNS_4 backend

Returns: mpv_table[n]
NULL

2.11.7 mpvideo_mode.c

The functions in the mpvideo_mode.c CSU are used to define MPV modes (panel orientations) and to put them into effect. These functions are:

- mpvideo_set_mode
- mpvideo_define_mode

2.11.7.1 mpvideo_set_mode

The mpvideo_set_mode function is used to put the desired resolution mode (panel orientation) into effect. This function is called to load a default mode when the MPV is initialized. During runtime, it is called if the Simulation Host sends a MSG_SUBSYS_MODE message to change the resolution mode.

The function call is **mpvideo_set_mode(backend, channel, mode)**, where:

backend is the backend id

channel is the channel number

mode is the identifier assigned by the Simulation Host to the desired mode; default modes are identified by 0 through 4

Resolution modes are defined using the MSG_DEFINE_TX_MODE message; some default modes are provided. Each mode's definition specifies the applicable MPV mode, graphics path orientation, vertical and horizontal screen resolution, and vertical and horizontal offset of the displayed image. The modes are loaded into the f0_mode_select[] array by the mpvideo_define_mode function at initialization time.

mpvideo_set_mode does the following:

- Checks to see if the new mode is different from the current mode set.
- Calls `mpvideo_get_object_addr` to get a pointer to the MPV object in the specified backend.
- Locates the outgoing (to Force) message buffer.
- Passes the request to the Force board by pushing a `MSG_F0_MODE_SELECT` message onto the outgoing buffer.

Called By: `mpvideo_boot`
 `msg_subsys_mode`

Routines Called: `mpvideo_get_object_addr`
 `mx2_push`
 `printf`

Parameters: `UNS_4` `backend`
 `UNS_2` `channel`
 `UNS_2` `mode`

Returns: `none`

2.11.7.2 `mpvideo_define_mode`

The `mpvideo_define_mode` function is used at initialization time to define all desired resolution modes of the MPV board. This information is stored for later use during runtime. The modes come from two sources: (1) default modes loaded by `mpvideo_boot`, and (2) modes defined by the Simulation Host using `MSG_DEFINE_TX_MODE` messages. Each backend can have different mode information.

The function call is `mpvideo_define_mode(backend, mode, mpv_mode, orient, i, j, ofi, ofj)`, where:

backend is the backend id

mode is an identifier assigned by the Simulation Host for the mode being defined; this label is used during runtime to put this mode into effect; the default modes are labeled 0 through 4

mpv_mode is 0 (single 640x480 channel), 1 (dual 320x240 channels replicated to 640x480), 2 (dual 640x256 channels), or 3 (single 640x480 channel replicated to 640x480)

orient is 0 (vertical orientation of the graphics path) or 1 (horizontal orientation of the graphics path); horizontal orientation is not currently supported

i is horizontal screen resolution (number of pixels to display per line)

j is the vertical screen resolution (number of pixels to display per column)

ofi is the desired horizontal offset of the displayed image (normal = 0)

ofj is the desired vertical offset of the displayed image (normal = 0)

For each MPV mode, `mpvideo_define_mode` does the following:

- If the `mpvideo_print` variable (checked by `status_mpvideo_print`) is TRUE, outputs all of the mode information to stdout.
- Adds the mode and its related data to the `f0_mode_select[]` array. This table is used to change modes during runtime.

Called By: `cig_config`
 `db_mcc_setup`
 `mpvideo_boot`

Routines Called: `printf`
 `status_mpvideo_print`

Parameters:	<code>UNS_4</code>	<code>backend</code>
	<code>INT_2</code>	<code>mode</code>
	<code>UNS_1</code>	<code>mpv_mode</code>
	<code>UNS_1</code>	<code>orient</code>
	<code>INT_2</code>	<code>i</code>
	<code>INT_2</code>	<code>j</code>
	<code>INT_2</code>	<code>ofi</code>
	<code>INT_2</code>	<code>ofj</code>

Returns: none

2.11.8 `mpvideo_pass_back.c`

The `mpvideo_pass_back` function gets the data for the `MSG_PASS_BACK` message for return to the Simulation Host. This message is used to return status information from an embedded subsystem (currently, only the 2-D overlay processor).

The function call is `mpvideo_pass_back(backend, pmsg_ptr)`, where:

backend is the backend id

pmsg_ptr is a pointer to the data for the `MSG_PASS_BACK` message

`mpvideo_pass_back` does the following:

- Calls `mpvideo_get_object_addr` to get a pointer to the MPV object in the specified backend.
- Verifies that the MPV's `pass_on_sent` flag is TRUE and the `data_buf_in` buffer (used to store `pass_on` data from the GSP) is not empty.
- Sets the *pmsg_ptr* to the `data_buf_in` buffer. The `msg_pass_back` function (in the Message Processing CSC) uses this pointer to build the `MSG_PASS_BACK` message.

The function returns 0 if successful. It returns EOF if it cannot get a pointer to the MPV object.

Called By: `msg_pass_back`

Routines Called: mpvideo_get_object_addr

Parameters: UNS_4 backend
 INT_2 **pmsg_ptr

Returns: 0
 EOF

2.11.9 mpvideo_pass_on.c

The mpvideo_pass_on function is used to send a message to the GSP on a specified backend. This function is called whenever the Simulation Host sends a MSG_PASS_ON message to pass runtime changes to the 2-D overlay processor (task2d).

The function call is mpvideo_pass_on(backend, channel, msg, size), where:

backend is the backend id

channel is the channel number; this determines which subsystem process receives the message; currently, the only implemented value is 1 (2-D overlay processor)

msg is a pointer to the MSG_PASS_ON message

size is the size of the message in bytes

mpvideo_pass_on does the following:

- Calls mpvideo_get_object_addr to get a pointer to the MPV object in the specified backend.
- If the *channel* is 1 (2-D overlay processor):
 - Moves the message data to the MPV's data_buf_out buffer.
 - Sets the MPV's io_req flag to TRUE.
 - Sets the MPV's pass_on_sent flag to TRUE.

The function returns 0 if successful. It returns EOF if it cannot get a pointer to the MPV object.

Called By: msg_pass_on

Routines Called: min
 mpvideo_get_object_addr

Parameters: UNS_4 backend
 UNS_4 channel
 INT_2 *msg
 INT_4 size

Returns: 0

EOF

2.11.10 mpvideo_print.c

The functions in the mpvideo_print.c CSU control whether or not optional status information is printed by various MPV Interface routines. These functions are:

- status_mpvideo_print
- toggle_mpvideo_print

2.11.10.1 status_mpvideo_print

The status_mpvideo_print function returns the current value of the boolean variable mpvideo_print. If this variable is TRUE, the calling routine outputs optional status information to stdout. If FALSE, the information is not displayed. The default is FALSE; the variable can be set to TRUE using toggle_mpvideo_print.

The function call is **status_mpvideo_print()**.

Called By:	bootforce mpvideo_boot mpvideo_define_mode mpvideo_load
------------	--

Routines Called:	none
------------------	------

Parameters:	none
-------------	------

Returns:	mpvideo_print
----------	---------------

2.11.10.2 toggle_mpvideo_print

The toggle_mpvideo_print function toggles the value of the boolean variable mpvideo_print. This variable controls whether or not various MPV Interface routines print optional status information.

The function call is **toggle_mpvideo_print()**.

This function is not currently used.

Called By:	none
------------	------

Routines Called:	none
------------------	------

Parameters: none

Returns: none

2.11.11 mpvideo_query.c (mpvideo_num_paths)

The mpvideo_num_paths function is used to find out how many graphics paths a backend uses or is going to use.

The function call is **mpvideo_num_paths(backend)**, where *backend* is the backend id. The function does the following:

- Calls mpvideo_get_object_addr to get a pointer to the MPV object in the specified backend.
- Calculates the number of graphics paths based on the display configuration and the MPV mode.

If successful, the function returns the number of paths (2 or 4). It returns EOF if it cannot get a pointer to the MPV.

Called By: backend_set_thermal

Routines Called: mpvideo_get_object_addr

Parameters: UNS_4 backend

Returns: 2
4
EOF

2.11.12 mpvideo_response.c

The mpvideo_response function checks for and processes responses passed back from the MPV I/O board (via the Force task). This function is used primarily to get the laser range (pixel depth) value if laser range processing has been requested by the Simulation Host.

The function call is **mpvideo_response(lm_blk, backend)**, where:

lm_blk is the number of load modules per side of a load module block; this value is used to scale the range if load module blocking (extended viewing range) is enabled
backend is the backend id

mpvideo_response does the following:

- Calls mpvideo_get_object_addr to get a pointer to the MPV object in the specified backend.

- Calls `mx2_peek` to preview the top message in the incoming (from Force) message buffer.
- Processes the message according to its type.
- Calls `mx2_skip` to remove the processed message.
- Repeats the process with the next message in the incoming buffer.

The following table lists the message types handled by `mpvideo_response`. The second column shows the purpose of the message (in *italics*), then summarizes the major steps performed by `mpvideo_response`.

Message from MPV	Processing by <code>mpvideo_response</code>
MSG_F1_ACKNOWLEDGE	<i>Acknowledgment from Force task.</i> If debug mode is enabled, copies message and outputs data to stdout.
MSG_F1_MPV_IOCT	<i>Illegal Opcode Trap Message from MPV.</i> Copies message; outputs error information to stdout; calls <code>syserr</code> to generate error message.
MSG_F1_MPV_LUT_TYPE	<i>Reports MPV type and lookup table type.</i> If debug mode is enabled, copies message and outputs data to stdout.
MSG_F1_MPV_MEMORY	<i>Returns data from MPV memory.</i> If debug mode is enabled, copies message and outputs confirmation message to stdout.
MSG_F1_PIXEL_ADDR	<i>Returns pixel data.</i> If debug mode is enabled, copies message and outputs data to stdout.
MSG_F1_PIXEL_DEPTH_RETURN	<i>Returns pixel depth value for laser range request.</i> Copies message; uses <code>FXT0881</code> to convert depth value to floating point (or sets depth to -1.0 if invalid); calls <code>msg_laser_return</code> to generate laser return message for Simulation Host.
MSG_F1_STATUS	<i>Reports MPV status.</i> If debug mode is enabled, copies message and outputs data to stdout.
MSG_F1_TEXT	<i>Returns text read from Force.</i> If debug mode is enabled, copies message and outputs data to stdout.

The function returns 1 if it processes the message successfully. It returns EOF if it cannot get a pointer to the MPV object.

Called By: `backend_response`

Routines Called: `FXT0881`
`mpvideo_get_object_addr`
`msg_laser_return`
`mx2_hwcopy`
`mx2_peek`

mx2_skip
printf
syserr

Parameters:

INT_4
UNS_4

lm_blk
backend

Returns:

EOF
1

2.12 Message Processing (/cig/libsrc/libmsg)

The functions in the Message Processing CSC are responsible for processing many of the messages passed between the CIG and the Simulation Host. These functions handle both incoming and outgoing messages, during system configuration as well as during the simulation. Each Message Processing function handles a specific message type.

Most incoming messages are received by `process_a_msg` (in `simulation.c` in the Real-Time Processing CSC), which determines the message type and then calls the appropriate Message Processing function. The Message Processing function may then call a function in another CSC (e.g., Viewport Configuration or Backend Processing) to process the request.

The Message Processing functions do not process all incoming messages. In many cases, `process_a_msg` calls the appropriate update function directly or, for messages to be sent to Ballistics, pushes the message onto the Ballistics message queue. Similarly, messages to be returned to the Simulation Host may be generated by functions in other CSCs.

This CSC also contains the functions used by the Host Interface Manager CSC to print messages if debug message printing is enabled. This feature, which can be enabled through Gossip, lets the user print or display the contents of all or selected message types for testing or debugging purposes.

Figure 2-16 identifies the CSUs in the Message Processing CSC. These CSUs are described in this section.

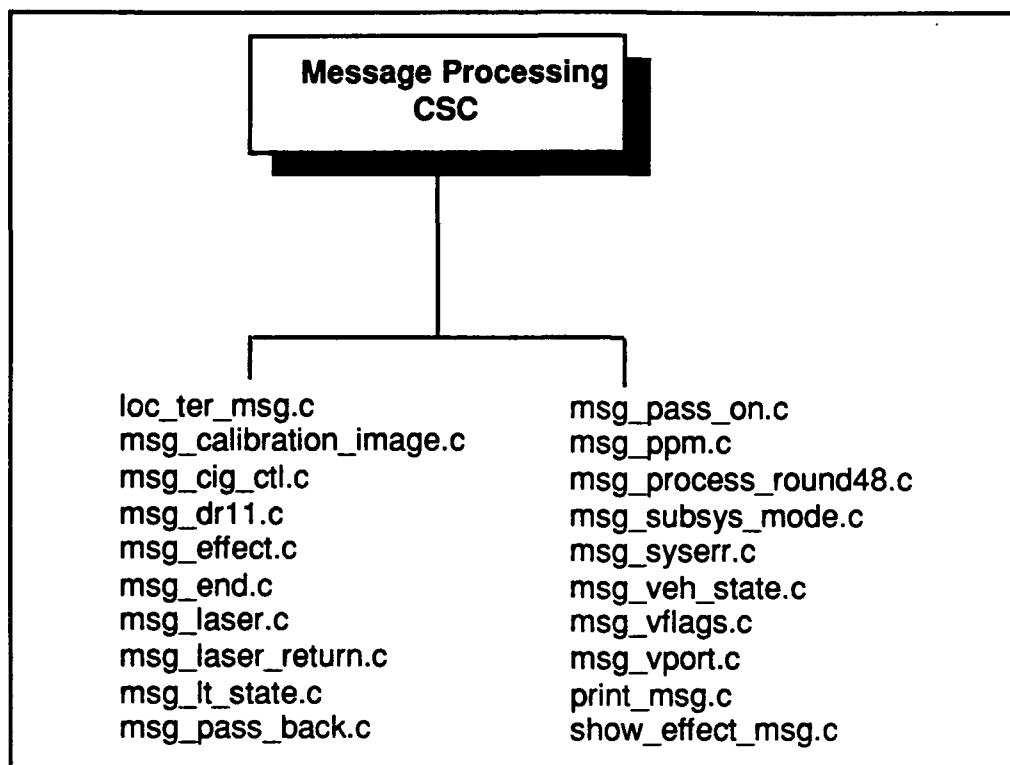


Figure 2-16. Message Processing CSUs

2.12.1 loc_ter_msg.c

The `loc_ter_msg` function processes the `MSG_LT_PIECE` message. This is a CIG-to-SIM message that provides a detailed description of the polygons and bounding volumes contained in the terrain around the simulated vehicle. Due to its size, the data is divided into several pieces and delivered as separate messages. Collectively, all of the `MSG_LT_PIECE` messages give the Simulation Host the data it uses to determine the position and orientation of the simulated vehicle, and to calculate vehicle dynamics.

By default, a local terrain message is sent to the Simulation Host every 32 frames. This interval can be changed using the `MSG_DR11_PKT_SIZE` or `MSG_LT_STATE` message. The `loc_ter_msg` function is called every frame, and it determines whether or not it is time to send a local terrain message.

The data for the `MSG_LT_PIECE` messages is generated by the `local_terrain` function in the Real-Time Processing CSC. `local_terrain` creates a `MSG_LOCAL_TERRAIN` message based on the four grids surrounding the simulated vehicle. `loc_ter_msg` then divides the data in the `MSG_LOCAL_TERRAIN` message into a series of smaller `MSG_LT_PIECE` messages, and returns the message to the Simulation Host.

The function call is `loc_ter_msg()`. `loc_ter_msg` does the following:

- Checks to see if the frame count since the last local terrain message was sent equals or exceeds the local terrain interval. This indicates that it is time to send another local terrain message.

- Calculates the number of pieces (the total size of the MSG_LOCAL_TERRAIN message divided by the local terrain chunk size, plus 1).
- Builds the MSG_LT_PIECE message for each piece and puts it in the outgoing message buffer.

Called By: exchange_dr11_data_sim
 exchange_enet_data_sim
 exchange_flea_data
 exchange_mpv_data_sim
 exchange_scsi_data_sim
 exchange_socket_data_sim

Routines Called: printf

Parameters: none

Returns: none

2.12.2 msg_calibration_image.c

The msg_calibration_image function processes the MSG_CALIBRATION_IMAGE message. This message is sent by the Simulation Host to control the display of monitor calibration images.

The function call is msg_calibration_image(msg_P), where msg_P is a pointer to the MSG_CALIBRATION_IMAGE message.

msg_calibration_image does the following:

- Verifies that the backend and image identifiers specified in the message are valid.
- Determines which backend is being updated.
- Sets a pointer to the calibration overlay area in the backend's AAM.
- Based on the image number specified in the message, proceeds as follows:
 - Image = 0 (turn images off)**
 - Sets pcal_flag in global memory to FALSE.
 - Image = 1 (display offset image)**
 - Allocates memory to create a temporary overlay.
 - Calls make_cal_matrices to generate the overlay's matrices.
 - Calls make_cal_patterns to generate the polygons, triangles, and other patterns displayed.
 - Copies the temporary overlay to active area memory.
 - Frees the memory used for the temporary overlay.
 - Sets pcal_flag to TRUE.
 - Image = 2 (display color image)**
 - Allocates memory to create a temporary overlay.
 - Calls make_cal_matrices to generate the overlay's matrices.
 - Calls gos_polys to create the overlay's polygons.
 - Copies the temporary overlay to active area memory.
 - Frees the memory used for the temporary overlay.

- Sets `pcal_flag` to TRUE.
- Image = 3 (display BBN logo)**
- Allocates memory to create a temporary overlay.
- Calls `make_cal_matrices` to generate the overlay's matrices.
- Calls `make_bbn_logo` to generate the patterns that form the logo display.
- Copies the temporary overlay to active area memory.
- Frees the memory used for the temporary overlay.
- Sets `pcal_flag` to TRUE.

The function outputs an error to stdout if there is insufficient memory for the overlay.

Called By: `process_a_msg`

Routines Called: `bcopy`
`free`
`GLOB`
`gos_polys`
`make_bbn_logo`
`make_cal_matrices`
`make_cal_patterns`
`malloc`
`printf`
`VME_TO_VMX`

Parameters: `MSG_CALIBRATION_IMAGE` `*msg_P`

Returns: `none`

2.12.3 `msg_cig_ctl.c`

The `msg_cig_ctl` function processes the `MSG_CIG_CTL` message if it is received during a simulation. This message is sent by the Simulation Host to stop the simulation. (The only valid state changes during a simulation are `C_STOP` and `C_NULL`.) `msg_cig_ctl` initializes the configuration tree, resets Ballistics, resets the backend processors, closes the database, and initializes the frame count.

The function call is `msg_cig_ctl(msgp, state)`, where:

msgp is a pointer to the `MSG_CIG_CTL` message
state is the current state of the CIG

`msg_cig_ctl` does the following:

- Reads the new state requested by the Simulation Host.
- If the new state is `C_STOP`:
 - Calls `vpt_tree_free` to free the viewport configuration tree.
 - Calls `sim_bal_reset` to reset Ballistics.
 - Calls `backend_reset` to reset the backend processors.

- If the old (current) state is C_SIMULATION:
 - * Calls sysrup_off to disable interrupts.
 - * If running in AGPT mode (a non-standard CIG mode), calls sim_state_init and sets agpt_mode to 1.
- Calls close_db to close the terrain database.
- Calls vpt_tree_init to initialize the viewport configuration tree.
- Sets the frame count to 0.
- Sets the new state to the state specified in the message (C_STOP).
- If the new state is C_NULL:
 - Sets the new state to C_SIMULATION (i.e., no state change).
- If any other state is specified:
 - Calls syserr to generate an illegal state transition message.
 - Sets the new state to C_SIMULATION (i.e., no state change).
- If real-time software timing has been enabled through Gossip (i.e., if rtsw_timing_flag is TRUE), computes the processing time for the message and updates the worst time if applicable.

The function returns the new state of the CIG as *new_state*.

Called By: process_a_msg

Routines Called: backend_reset
close_db
GLOB
printf (in debug mode only)
read_watch
sim_bal_reset
sim_state_init
syserr
sysrup_off
vpt_tree_free
vpt_tree_init

Parameters: MSG_CIG_CTL *msgp
INT_2 state

Returns: new_state

2.12.4 msg_dr11.c (msg_dr11_pkt_size)

The msg_dr11_pkt_size function processes the MSG_DR11_PKT_SIZE message. This message is sent by the Simulation Host to set the following:

- Simulation Host exchange packet size.
- CIG exchange packet size.
- Local terrain piece (chunk) size.
- Local terrain frame interval.
- CIG hardware type (always 0 for GTs).

The function call is `msg_dr11_pkt_size(msgp)`, where *msgp* is a pointer to the MSG_DR11_PKT_SIZE message.

The function does the following:

- Outputs the message contents to stdout.
- Sets the applicable global variables with the new packet sizes.
- Verifies that the local terrain chunk size and interval are above the minimum values allowed, then sets the applicable global variables with the new values.
- Sets the global variable for the CIG hardware type to the new value.

Called By: `process_a_msg`

Routines Called: `GLOB`
 `printf`

Parameters: `MSG_DR11_PKT_SIZE` **msgp*

Returns: none

2.12.5 `msg_effect.c` (`msg_show_effect`)

The `msg_show_effect` function process the MSG_SHOW_EFFECT message. This message is sent from the Simulation Host to tell the CIG to display a special effect at a specified location.

The function call is `msg_show_effect(msgp, pdbname)`, where:

msgp is a pointer to the MSG_SHOW_EFFECT message
pdbname is the primary database control block

`msg_show_effect` does the following:

- Increments the active effects counter.
- Verifies that there is room for another active effect.
- If the maximum number of active effects has already been reached:
 - Decrements the active effects counter.
 - Outputs an error to stdout.
 - Calls `msg_syserr` to generate a MSG_SYS_ERROR message for return to the Simulation Host.
- If there is room for another active effect:
 - Calls `show_effect_msg` to add the effect to the active effects table.
 - If a second AAM exists, calls `show_effect_msg` to add the effect to its active effects table.

Called By: `process_a_msg`

Routines Called:	msg_syerr printf show_effect_msg VME_TO_VMX	
Parameters:	MSG_SHOW_EFFECT DB_INFO	*msgp *pdbname
Returns:	none	

2.12.6 msg_end.c

The functions in the msg_end.c CSU handle all of the processing that must occur at the end of a frame (when the Simulation Host sends a MSG_END message). These functions are:

- msg_end
- _downcount_effects
- _display_lights
- _move_load_module_stp_to_quad_buffer
- _update_second_active_area_memory
- _pend_on_frame_interrupt
- _process_agl
- _set_up_for_next_frame
- _handle_request_local_terrain
- _database_disable
- _handle_point_lights
- _reset_model_pointers
- _copy_reconfigurable_viewports_section

msg_end is the driving function. The other routines are called by msg_end to perform a specific task.

2.12.6.1 msg_end

The msg_end function processes the MSG_END message. This message is sent from the Simulation Host to signal the end of the contents of a packet buffer.

The function call is **msg_end(state, curr_sim_buf, pdbname, ppdynl, pptanks, ppmodels, ppeffects)**, where:

state is the current state of the CIG
curr_sim_buf is the buffer currently being used by the simulation
pdbname is as a pointer to the primary database control block
ppdynl is a pointer to the structure containing the current state of all dynamic models
pptanks is a pointer to the tank table
ppmodels is a pointer to the model table
ppeffects is a pointer to the dynamic effects table

msg_end does the following:

- Calls `backend_send_req` to initiate backend processing of messages queued up during the previous frame.
- If the system is running in AGPT mode (a non-standard mode), calls `sc_accept`, `rt_pend`, and `sim_receive` to check for any stacked up messages that still need to be processed this frame.
- Calls `_downcount_effects` to decrement the count for multiple-frame special effects.
- Calls `_display_lights` to display the correct LEDs.
- Calls `_move_load_module_stp_to_quad_buffer` to move the load module state table pointers to the quad buffer.
- Calls `_update_second_active_area_memory` to copy the new data to AAM2.
- If cloud processing is enabled, calls `cloud_mgmt`.
- Calls `sio_tick` to write all serial I/O data queued for the current frame to the specified devices.
- Calls `cigsimio_frame_end` to trigger the display or printing of messages from the previous frame (if debug message display or message recording is enabled).
- Calls `_pend_on_frame_interrupt` to wait for the next frame interrupt.
- Calls `sim_bal_process_msg` to process all messages returned from Ballistics.
- Calls `msg_pass_back` to generate any required MSG_PASS_BACK messages.
- Calls `backend_response` to obtain laser depth return data for each AAM.
- Calls `_process_agl` to generate the MSG_AGL message, if requested.
- Calls `_set_up_for_next_frame` to reset the state table to prepare for the next frame.
- Calls `_handle_request_local_terrain` to initiate the local terrain message.
- Calls `_database_disable` to initiate database management (`rowcol_rd`).
- Increments the `frame_count`.
- Calls `_handle_point_lights` to provide a time parameter for point lights.
- Calls `_reset_model_pointers` to reset all model table pointers.
- Calls `_copy_reconfigurable_viewports_section` to copy the current reconfigurable viewport parameters to the next frame's double buffer.
- Calls `sim_bal_process_tracer` to process all tracer (round position) messages returned by Ballistics.
- Calls `sim_bal_agl_wanted` to send a MSG_B0_TRAJ_CHORD message to report the simulated vehicle's current altitude.
- If real-time software timing has been enabled through Gossip (i.e., if `rtsw_timing_flag` is TRUE), computes the processing time for the message and updates the worst time if applicable.

Called By: `process_a_msg`

Routines Called:

- `_copy_reconfigurable_viewports_section`
- `_database_disable`
- `_display_lights`
- `_downcount_effects`
- `_handle_point_lights`
- `_handle_request_local_terrain`
- `_move_load_module_stp_to_quad_buffer`
- `_pend_on_frame_interrupt`
- `_process_agl`
- `_reset_model_pointers`
- `_set_up_for_next_frame`
- `_update_second_active_area_memory`
- `backend_response`
- `backend_send_req`

cigsimio_frame_end
 cloud_mgmt
 msg_pass_back
 printf (in debug mode only)
 read_watch
 rt_pend
 sc_accept
 sim_bal_agl_wanted
 sim_bal_process_msg
 sim_bal_process_tracer
 sim_receive
 sio_tick

Parameters:	INT_2 INT_2 DB_INFO INT_4 TANK OMODEL SHOW_EFF	state *curr_sim_buf *pdbase **ppdynl **pptanks **ppmodels **ppeffects
-------------	--	---

Returns: none

2.12.6.2 _downcount_effects

The `_downcount_effects` function counts down multiple-frame special effects. This function is called at the end of each frame.

The function call is `_downcount_effects(pdbase)`, where *pdbase* is a pointer to the primary database control block.

`_downcount_effects` calls `effect_downcount` to decrement the frame count for each effect. If two AAMs are defined, the function calls `effect_downcount` for each.

Called By:	msg_end	
Routines Called:	effect_downcount printf (in debug mode only)	
Parameters:	DB_INFO	*pdbase
Returns:	none	

2.12.6.3 `_display_lights`

The `_display_lights` function is used to display LED lights on the EVC. This function is called at the end of every frame.

The function call is `_display_lights()`. The function determines which lights/values to display: model count, frame count, local terrain count, or overload. It then calls `set_leds` to set the LED display.

`_display_lights` determines which lights to display by examining the `display_lights[]` array. This array contains a flag that enables/disables each counter. The flags in `display_lights[]` can be changed using the I ("set display lights flags") option on the Gossip System menu.

Called By: `msg_end`

Routines Called: `set_leds`

Parameters: `none`

Returns: `none`

2.12.6.4 `_move_load_module_stp_to_quad_buffer`

The `_move_load_module_stp_to_quad_buffer` function moves the load module state table pointers (STP) to the quad buffer. This function is called at the end of every frame.

The function call is `_move_load_module_stp_to_quad_buffer(pdbase, pdynl)`, where:

pdbase is a pointer to the primary database control block

pdynl is a pointer to the structure containing the current state of all dynamic models

The function does the following:

- If the current double buffer (`db_G`) is DB1:
 - Copies the load module state table in DB1 (size = `LM_COUNT * 4`) to *pdynl*.
- If real-time software timing has been enabled through Gossip (i.e., if `rtsw_timing_flag` is TRUE), computes the processing time and updates the worst time if applicable.

Called By: `msg_end`

Routines Called: `blcopy`
`printf` (in debug mode only)

read_watch

Parameters:	DB_INFO	*pdbase
	INT_4	*pdynl

Returns: none

2.12.6.5 _update_second_active_area_memory

The `_update_second_active_area_memory` function is responsible for copying new data to the second AAM, if one is defined. This function is called at the end of every frame.

The function call is `_update_second_active_area_memory(pdbase, ptanks, pmodels, peffects)`, where:

pdbase is a pointer to the primary database control block
ptanks is a pointer to the tank table
pmodels is a pointer to the model table
peffects is a pointer to the dynamic effects table

Called By: msg_end

Routines Called:	AAM1_TO_AAM2	
	blcopy	
	printf	(in debug mode only)
	return_aam_ptr	

Parameters:	DB_INFO	*pdbase
	TANK	*ptanks
	OMODEL	*pmodels
	SHOW_EFF	*peffects

Returns: none

2.12.6.6 _pend_on_frame_interrupt

The `_pend_on_frame_interrupt` function waits for the next frame interrupt. This function is called when a new simulation is initialized and at the end of every frame.

The function call is `_pend_on_frame_interrupt(state)`, where *state* is the current state of the CIG. `_pend_on_frame_interrupt` does the following:

- If the state is C_SIMULATION, and the SIMULATION_MB mailbox contains a message (indicating that an interrupt has occurred), calls `syserr` to generate a system overload error message.
- Waits for a message to be posted to the SIMULATION_MB mailbox.

- Called By:** init_simulation
msg_end

Parameters: INT_2 state

2.12.6.7 `_process_agl`

The function call is `_process_agl()`. The function does the following:

- Called By:** msg_end

Parameters: none

Returns: `none`

2.12.6.8 _set_up_for_next_frame

The `_set_up_for_next_frame` function resets the state table and manipulates various pointers and variables necessary to prepare for the next frame. This function is called at the end of every frame.

The function call is `_set_up_for_next_frame(state)`, where *state* is the current state of the CIG.

The function does the following:

- If real-time software timing has been enabled through Gossip (i.e., if `rtsw_timing_flag` is TRUE), computes the processing time and updates the worst time if applicable.
- Calls `cigsimio_msg_out` to write the messages in the outgoing packet to a buffer (if debug message display or message recording is enabled).
- Calls the appropriate host interface function (using the `*exchange_data_sim` function pointer) to exchange packet buffers.
- If real-time software timing has been enabled through Gossip (i.e., if `rtsw_timing_flag` is TRUE), computes the processing time and updates the worst time if applicable.
- If single-step mode is enabled, sets `dr11_msg` to TRUE.

Called By: msg_end

Routines Called: *exchange_data_sim
 cigsimio_msg_out
 read_watch

Parameters: INT 2 state

Returns: `none`

2.12.6.9 handle_request_local_terrain

The `_handle_request_local_terrain` function initiates the local terrain message process if it is time to return a `MSG_LOCAL_TERRAIN` message to the Simulation Host. This function is called at the end of every frame.

The function call is `_handle_request_local_terrain(pdbase)`, where *pdbase* is a pointer to the primary database control block.

The function does the following:

- Checks to see if local terrain processing is enabled.
- Checks to see if the required number of frames have passed since the last message.

- If both conditions are true, posts a message to the LOCAL_TERRAIN_MB mailbox to initiate preparation of a local terrain message.

Called By: msg_end

Routines Called: sc_post

Parameters: DB_INFO *pdbase

Returns: none

2.12.6.10 _database_disable

The _database_disable function initiates database management by posting a mailbox message to invoke rowcol_rd. This function is called at the end of every frame. It invokes rowcol_rd only if the specified frame interval has passed since database management was last performed.

The function call is _database_disable(). The function does the following:

- Makes sure the database_disable_wanted variable is FALSE. If TRUE, no database management is performed after the initial database is loaded. This variable, initialized to FALSE, can be set TRUE by specifying the "c" (constrain) argument on the start-up command line.
- Checks to see if the current frame count is greater than the database's init_freq (initial frequency), indicating that the rowcol_rd task is beyond its initial starting point. The init_freq, set up by config_database in the Real-Time Processing CSC, is set at 16 frames for even-numbered backends and 32 frames for odd-numbered backends.
- Checks to see if the specified number of frames (32 by default) have passed since the last call to rowcol_rd.
- If all of the above criteria are met, posts a message to the database's mailbox to start the rowcol_rd task.
- Repeats this process for each instance of the rowcol_rd task (one per backend).

Called By: msg_end

Routines Called: sc_post

Parameters: none

Returns: none

2.12.6.11 `_handle_point_lights`

The `_handle_point_lights` function provides a time parameter for handling point lights standardized on a 30-Hertz frame rate.. This function is called at the end of every frame.

Point lights are light sources that can be defined from a specific location in xyz world space. This is different from directional lighting, which is defined from a certain direction and originates at infinity.

Point lights are specified in the MSG_VIEW_FLAGS message. For each point light, 0=off, 1=low intensity, and 2=high intensity. The 12 light groups are:

A = spare	G = spare
B = spare	H = VASI
C = obstruction lights	I = approach strobes
D = hazard beacon	J = approach lights
E = military airport beacon	K = taxiway lights
F = commercial airport beacon	L = runway lights

The function call is `_handle_point_lights()`. The function sets the current time in both AAMs based on the current frame rate/count.

Called By: `msg_end`

Routines Called: `AAM1_TO_AAM2`

Parameters: `none`

Returns: `none`

2.12.6.12 `_reset_model_pointers`

The `_reset_model_pointers` function resets all model table pointers. This function is called at the end of every frame.

The function call is `_reset_model_pointers(curr_sim_buf, pdbase, ppdynl, pptanks, ppmodels, ppeffects)`, where:

curr_sim_buf is the buffer currently being used by the simulation
pdbase is as a pointer to the primary database control block
ppdynl is a pointer to the structure containing the current state of all dynamic models
pptanks is a pointer to the tank table
ppmodels is a pointer to the model table
ppeffects is a pointer to the dynamic effects table

`_reset_model_pointers` does the following:

- Resets the model counter to 0.
- Calls `get_dtp_bank` to determine which double buffer to update.
- Sets the model table pointers based on which buffer section the simulation is using.
- Sets the next tank, next model, and next effect pointers.
- Resets the dynamic model return addresses to the static model table.
- Resets the dynamic pre- and post-processed model pointers.
- If a second AAM is present, resets its model pointers also.

Called By: `msg_end`

Routines Called: `get_dtp_bank`

Parameters:	<code>INT_2</code>	<code>*curr_sim_buf</code>
	<code>DB_INFO</code>	<code>*pdbase</code>
	<code>INT_4</code>	<code>**ppdynl</code>
	<code>TANK</code>	<code>**pptanks</code>
	<code>OMODEL</code>	<code>**ppmodels</code>
	<code>SHOW_EFF</code>	<code>**ppeffects</code>

Returns: `none`

2.12.6.13 `_copy_reconfigurable_viewports_section`

The `_copy_reconfigurable_viewports_section` function copies the previous frame's reconfigurable viewports portion of the double buffer into the new frame's double buffer. This function is called at the end of every frame.

The function call is `_copy_reconfigurable_viewports_section()`. The function does the following:

- Resets the dynamic pre- and post-processed model pointers.
- Copies the reconfigurable data from the working double buffer to the other buffer.

Called By: `msg_end`

Routines Called: `blcopy`
`return_aam_ptr`

Parameters: `none`

Returns: `none`

2.12.7 msg_laser.c (msg_laser_request_range)

The `msg_laser_request_range` function processes the `MSG_REQUEST_LASER_RANGE` message. This message is sent by the Simulation Host to request laser range information for a viewport. The message specifies the screen position (row and column pixels) from which to return the range value each frame. Each frame, the CIG returns the distance from the viewpoint to the object within the requested pixel. For T backends (which do not have a Force board), a hard-wired pixel is used to determine range.

The function call is `msg_laser_request_range(msgp, state)`, where:

msgp is a pointer to the `MSG_REQUEST_LASER_RANGE` message
state is the current state of the CIG

`msg_laser_request_range` does the following:

- Calls `backend_laser_request_range` to enable laser request processing.
- If the request cannot be processed, calls `syserr` to generate an error message for return to the Simulation Host.
- If real-time software timing has been enabled through Gossip (i.e., if `rtsw_timing_flag` is `TRUE`), computes processing time for message and updates worst time if applicable.

Called By: `process_a_msg`

Routines Called: `backend_laser_request_range`
`read_watch`
`syserr`

Parameters: `MSG_REQUEST_LASER_RANGE` `*msgp`
`INT_2` `state`

Returns: `none`

2.12.8 msg_laser_return.c

The `msg_laser_return` function builds the `MSG_LASER_RETURN` message. This message is sent by the CIG to the Simulation Host to provide laser return information for a specified pixel within the screen. The message is sent every frame if the Simulation Host has requested laser return data with the `MSG_REQUEST_LASER_RANGE` message.

The function call is `msg_laser_return(lm_blk, subsys, channel, id, range)`, where:

lm_blk is the number of load modules per side of a load module block; this value is used to scale the range if an extended viewing range is enabled
subsys is the id of the subsystem (backend) returning the data

channel is channel to which the range value applies
id is the identifier assigned to this request in the corresponding
 MSG_REQUEST_LASER_RANGE message
range is the range of the object (in meters) in the selected pixel

msg_laser_return uses the information provided by backend_response (for a T backend) or mpvideo_response (for a TX backend) to build the MSG_LASER_RETURN message.

Called By: backend_response
 mpvideo_response

Routines Called: panic

Parameters:	INT_4	lm_blk
	UNS_4	subsys
	INT_2	channel
	INT_2	id
	REAL_4	range

Returns: none

2.12.9 msg_lt_state.c (lt_state)

The lt_state function processes the MSG_LT_STATE message. This message can be sent by the Simulation Host to enable or disable local terrain processing, and to change the local terrain interval and/or chunk size. The message may be sent during system configuration or runtime.

If local terrain processing is enabled, the CIG sends MSG_LOCAL_TERRAIN messages to the Simulation Host at the specified interval (frame count). Due to its large size, the message is divided into multiple MSG_LT_PIECE messages, each the specified "chunk" size.

The function call is lt_state(msgp), where msgp is a pointer to the MSG_LT_STATE message.

lt_state does the following:

- Sets the local_terrain_wanted variable to the value specified in the message.
- If local terrain processing is enabled, sets the new interval specified in the message.
- Verifies that the new chunk size is at least 50 bytes less than the current outgoing packet size.
 - If it is, sets the new chunk size.
 - If it is not, generates an error message.

Called By: cig_config
 process_a_msg

Routines Called: printf
 syserr

Parameters: MSG_LT_STATE *msgp

Returns: none

2.12.10 msg_pass_back.c

The msg_pass_back function processes the MSG_PASS_BACK message. This message is sent from the CIG to the Simulation Host to return information from an embedded processor. This function is called at the end of every frame.

The MSG_PASS_BACK message applies to TX backends only. At the current time, all pass_back messages are generated from the 2-D task running on the MPV board.

The function call is msg_pass_back(). The function obtains the data from mpvideo_pass_back, then builds the message and puts it in the outgoing message buffer.

Called By: msg_end

Routines Called: mpvideo_pass_back

Parameters: none

Returns: none

2.12.11 msg_pass_on.c

The msg_pass_on function processes the MSG_PASS_ON message. This message is sent by the Simulation Host to tell the CIG to pass the message on to an embedded subsystem in the CIG. Currently, the only subsystem that receives pass_on messages is the 2-D task on the MPV board.

The function call is msg_pass_on(msgp, state), where:

msgp is a pointer to the MSG_PASS_ON message
state is the current state of the CIG

msg_pass_on does the following:

- Calls mpvideo_pass_on to put the message into the MPV's message buffer.
- If the MPV board cannot be found, calls syserr to generate an error message for return to the Simulation Host.

- If real-time software timing has been enabled through Gossip (i.e., if `rtsw_timing_flag` is TRUE), computes processing time for message and updates worst time if applicable.

Called By: `process_a_msg`

Routines Called: `mpvideo_pass_on`
`read_watch`
`syserr`

Parameters: `MSG_PASS_ON` `*msgp`
`INT_2` `state`

Returns: `none`

2.12.12 `msg_ppm.c`

The functions in the `msg_ppm.c` CSU are used to process messages that affect the Pixel Processor Memory (PPM) board. These functions are:

- `msg_ppm_display_mode`
- `msg_ppm_display_offset`
- `msg_ppm_pixel_location`
- `msg_ppm_pixel_state`

The PPM stores pixel color, weight, and depth values according to screen coordinates and pixel depth. The PPM eliminates hidden surfaces, color-averages pixels to display transparent pixels, and uses depth priority to display one type of pixel over another. To display one screen of pixel data while constructing another, the PPM double buffers the pixels' color and weight data.

Messages from the Simulation Host can specify which channels are displayed and which bank of the final color lookup table is used for sky color.

These functions apply to T backends only. Changes for the PPM are downloaded via the ESIFA board using the ESIFA Interface routines.

2.12.12.1 `msg_ppm_display_mode`

The `msg_ppm_display_mode` function processes the `MSG_PPM_DISPLAY_MODE` message. This message can be sent by the Simulation Host to set the display size (320 x 200, 320 x 128, 200 x 200, or 200 x 128).

The function call is `msg_ppm_display_mode(msg_P)`, where `msg_P` is a pointer to the `MSG_PPM_DISPLAY_MODE` message.

The function does the following:

- Calls ppm_get_data to get a pointer to the PPM object.
- Sets the new display mode and screen size.
- Calls esifa_queue_data to queue the data in ESIFA RAM.
- Calls esifa_queue_download to queue the data for download to the ESIFA.

Called By: db_mcc_setup
process_a_msg
upstart

Routines Called: ppm_get_data
esifa_queue_data
esifa_queue_download

Parameters: MSG_PPM_DISPLAY_MODE *msg_P

Returns: none

2.12.12.2 msg_ppm_display_offset

The msg_ppm_display_offset function processes the MSG_PPM_DISPLAY_OFFSET message. This message can be sent by the Simulation Host to set new horizontal and vertical display offset values.

The function call is msg_ppm_display_offset (msg_P), where msg_P is a pointer to the MSG_PPM_DISPLAY_OFFSET message.

The function does the following:

- Calls ppm_get_data to get a pointer to the PPM object.
- Sets the new horizontal and vertical display offsets.
- Calls esifa_queue_data to queue the data in ESIFA RAM.
- Calls esifa_queue_download to queue the data for download to the ESIFA.

Called By: db_mcc_setup
process_a_msg
upstart

Routines Called: ppm_get_data
esifa_queue_data
esifa_queue_download

Parameters: MSG_PPM_DISPLAY_OFFSET *msg_P

Returns: none

2.12.12.3 msg_ppm_pixel_location

The `msg_ppm_pixel_location` function processes the `MSG_PPM_PIXEL_LOCATION` message. This message can be sent by the Simulation Host to specify a new starting pixel location (horizontal and vertical coordinates) on the display screen.

The function call is `msg_ppm_pixel_location (msg_P)`, where `msg_P` is a pointer to the `MSG_PPM_PIXEL_LOCATION` message.

The function does the following:

- Calls `ppm_get_data` to get a pointer to the PPM object.
- Sets the new pixel location.
- Calls `esifa_queue_data` to queue the data in ESIFA RAM.
- Calls `esifa_queue_download` to queue the data for download to the ESIFA.

Called By: `db_mcc_setup`
 `process_a_msg`
 `upstart`

Routines Called: `ppm_get_data`
 `esifa_queue_data`
 `esifa_queue_download`

Parameters: `MSG_PPM_PIXEL_LOCATION` `*msg_P`

Returns: `none`

2.12.12.4 msg_ppm_pixel_state

The `msg_ppm_pixel_state` function processes the `MSG_PPM_PIXEL_STATE` message. This message can be sent by the Simulation Host to toggle the PPM pixel state on or off.

The function call is `msg_ppm_pixel_state (msg_P)`, where `msg_P` is a pointer to the `MSG_PPM_PIXEL_STATE` message.

The function does the following:

- Calls `ppm_get_data` to get a pointer to the PPM object.
- Sets the board address based on the new state.
- Calls `esifa_queue_data` to queue the data in ESIFA RAM.
- Calls `esifa_queue_download` to queue the data for download to the ESIFA.

Called By: `cal`
 `db_mcc_setup`
 `process_a_msg`

upstart

Routines Called: ppm_get_data
 esifa_queue_data
 esifa_queue_download

Parameters: MSG_PPM_PIXEL_STATE *msg_P

Returns: none

2.12.13 msg_process_round48.c

The msg_process_round48 function is not used by the standard GT100 system.

2.12.14 msg_subsys_mode.c

The msg_subsys_mode function processes the MSG_SUBSYS_MODE message. This message is sent by the Simulation Host to change parameters specific to a subsystem (backend). Specifically, the message can be used to do the following:

- Change the color lookup table displayed by all viewports in the subsystem.
- Change the fade value displayed by all viewports in the subsystem.
- Enable or disable calibration mode (T subsystems only).
- Set or change the TX subsystem mode. (Modes are defined using the MSG_DEFINE_TX_MODE message.)

The function call is msg_subsys_mode(msgp, state), where:

msgp is a pointer to the MSG_SUBSYS_MODE message
state is the current state of the CIG

msg_subsys_mode does the following:

- Validates the subsystem and channel identifiers in the message; calls syserr to generate an error message if either parameter is invalid.
- Calls esifa_set_special to set any special-use parameters specified in the message.
- Calls mpvideo_set_mode to set the new video control registers.
- Calls backend_set_color to set the new color table.
- Validates the new fade value against the range allowed by the color table; calls syserr to generate an error message if the value is invalid.
- Calls esifa_set_fade to set the new fade value.

Called By: process_a_msg

Routines Called: backend_set_color
 esifa_set_fade
 esifa_set_special

mpvideo_set_mode
syserr

Parameters: MSG_SUBSYS_MODE *msgp
INT_2 state

Returns: none

2.12.15 msg_syserr.c

The `msg_syserr` function builds a `MSG_SYS_ERROR` message for return to the Simulation Host. This message is sent every frame, whether or not an error has occurred. `msg_syserr` is called only if an error is to be reported by a Message Processing function, usually because the CIG could not process a message. (Note that several Message Processing functions call `syserr` to generate the error message.)

The function call is `msg_syserr(error, data, flush_flag, console_msg)`, where:

error is a code identifying the error type
data is a short text description of the error
flush_flag is TRUE if the input message buffer is to be flushed (i.e., any remaining messages are not to be processed)
console_msg is any additional text to be displayed to the console operator

`msg_syserr` does the following:

- Builds the outgoing message and header, and adds them to the outgoing message packet.
- If *flush_flag* is TRUE, sets the input message pointer to the end of the buffer, and outputs the *error*, *data*, and *console_msg* to stdout.

Called By: msg_otherveh_state
msg_show_effect
msg_staticveh_rem
msg_staticveh_state

Routines Called: printf

Parameters: INT_2 error
INT_2 data
INT_4 flush_flag
char *console_msg

Returns: none

2.12.16 msg_veh_state.c

The msg_veh_state.c CSU contains functions that process the SIM-to-CIG messages related to static and dynamic vehicles in the simulation environment. These functions are:

- msg_otherveh_state
- msg_staticveh_state
- msg_staticveh_rem

2.12.16.1 msg_otherveh_state

The msg_otherveh_state function processes the MSG_OTHERVEH_STATE message. This message is sent by the Simulation Host to report the position of a dynamic vehicle. One MSG_OTHERVEH_STATE message must be sent every frame for every dynamic vehicle within viewing range, whether or not the vehicle has moved since the previous frame. (The state tables for dynamic vehicles are completely rebuilt each frame with the new vehicle positions.)

The function call is msg_otherveh_state(msgp, pdbase), where:

msgp is a pointer to the MSG_OTHERVEH_STATE message
pdbase is a pointer to the primary database control block

msg_otherveh_state does the following:

- If the model counter is 0, sets an internal error flag to FALSE.
- Increments the model counter.
- Calls otherveh_state to add the vehicle to the appropriate tables.
- If the dynamic vehicle limit has already been reached:
 - If the error flag is FALSE, calls msg_syserr to generate an outgoing error message, then sets the error flag to TRUE. (This prevents another error message from being generated for every additional model over the limit.)
 - Decrements the model counter.
- If a second AAM is present, calls otherveh_state to add the vehicle to its tables.

Called By: process_a_msg

Routines Called: msg_syserr
otherveh_state
VME_TO_VMX

Parameters: MSG_OTHERVEH_STATE *msgp
DB_INFO *pdbase

Returns: none

2.12.16.2 msg_staticveh_state

The `msg_staticveh_state` function processes the `MSG_STATICVEH_STATE` message. This message is sent by the Simulation Host to place a static (non-moving) vehicle or obstacle at a specified location within the viewing range of the simulated vehicle.

The function call is `msg_staticveh_state(msggp, pdbase)`, where:

msggp is a pointer to the `MSG_STATICVEH_STATE` message
pdbase is a pointer to the primary database control block

`msg_staticveh_state` does the following:

- Increments the static vehicle counter.
- Calls `staticveh_state` to add the vehicle to the appropriate tables.
- If the static vehicle limit has already been reached:
 - Calls `msg_syserr` to generate an outgoing error message.
 - Decrements the static vehicle counter.
- If a second AAM is present, calls `staticveh_state` to add the vehicle to its tables.

Called By: `process_a_msg`

Routines Called: `msg_syserr`
`staticveh_state`
`VME_TO_VMX`

Parameters: `MSG_STATICVEH_STATE` `*msggp`
`DB_INFO` `*pdbase`

Returns: `none`

2.12.16.3 msg_staticveh_rem

The `msg_staticveh_rem` function processes the `MSG_STATICVEH_REM` message. This message is sent by the Simulation Host to remove a static (non-moving) vehicle or obstacle from the display. A `MSG_STATICVEH_REM` message is also generated by the `_rowcol_rd` (database management) function, if it detects that a static vehicle is no longer within viewing range of the simulated vehicle.

The function call is `msg_staticveh_rem(msggp, pdbase)`, where:

msggp is a pointer to the `MSG_STATICVEH_REM` message
pdbase is a pointer to the primary database control block

`msg_staticveh_rem` does the following:

- Decrements the static vehicle counter.

- If the vehicle type is 0, calls `msg_syserr` to generate an outgoing error message. (This is a vehicle that `_rowcol_rd` has detected beyond the viewing range.)
- Calls `staticveh_remove` to delete the vehicle from the appropriate tables.
- If a second AAM is present, calls `staticveh_remove` to delete the vehicle from its tables.

Called By: `process_a_msg`

Routines Called: `msg_syserr`
`staticveh_remove`
`VME_TO_VMX`

Parameters: `MSG_STATICVEH_REM` `*msgp`
`DB_INFO` `*pdbase`

Returns: `none`

2.12.17 `msg_vflags.c` (`msg_view_flags`)

The `msg_view_flags` function processes the `MSG_VIEW_FLAGS` message. This message is sent by the Simulation Host to update the system view flags and the branch value array. System view flags are used to turn individual processing paths on and off. The branch value array determines which branch is taken in each conditional node in the viewport configuration tree.

The function call is `msg_view_flags (msgp)`, where `msgp` is a pointer to the `MSG_VIEW_FLAGS` message.

The function does the following:

- If the view flags have changed since the previous settings:
 - Makes a copy of the new view flags (for comparison next time).
 - Calls `backend_set_paths` to set the new flags.
- Calls `backend_set_branch` to set the new branch values.

Called By: `process_a_msg`

Routines Called: `backend_set_branch`
`backend_set_paths`

Parameters: `MSG_VIEW_FLAGS` `*msgp`

Returns: `none`

2.12.18 msg_vport.c

The msg_vport.c CSU contains functions that process runtime changes to viewport parameters. These functions are:

- msg_viewport_update
- msg_view_magnification
- msg_rot2x1_matrix
- msg_rts4x3_matrix
- msg_hprxyzs_matrix
- msg_translation
- msg_scale
- msg_1rotation
- msg_3rotations

Most of these functions call an update function in the Viewport Configuration CSC to put the new parameter into effect.

2.12.18.1 msg_viewport_update

The msg_viewport_update function processes the MSG_VIEWPORT_UPDATE message. This message is sent by the Simulation Host to turn a viewport on or off, or to change the viewport's mode (e.g., thermal white hot or thermal black hot).

The function call is **msg_viewport_update(msgp)**, where *msgp* is a pointer to the MSG_VIEWPORT_UPDATE message.

The function does the following:

- Determines the channel id based on the viewport id in the message.
- Calls backend_set_video to turn the viewport channel on or off.
- If the specified backend cannot be found, calls syserr to generate an outgoing error message.
- Calls backend_set_thermal to set the new viewport mode.

Called By: process_a_msg

Routines Called: backend_set_thermal
backend_set_video
syserr

Parameters: MSG_VIEWPORT_UPDATE *msgp

Returns: none

2.12.18.2 msg_view_magnification

The `msg_view_magnification` function processes the `MSG_VIEW_MAGNIFICATION` message. This message is sent by the Simulation Host to change a viewport's field-of-view values and/or level-of-detail multiplier.

The function call is `msg_view_magnification(msgp)`, where `msgp` is a pointer to the `MSG_VIEW_MAGNIFICATION` message. The function calls `vpt_update_fov_lod` to perform the change.

Called By:	<code>process_a_msg</code>	
Routines Called:	<code>vpt_update_fov_lod</code>	
Parameters:	<code>MSG_VIEW_MAGNIFICATION</code>	<code>*msgp</code>
Returns:	<code>none</code>	

2.12.18.3 msg_rot2x1_matrix

The `msg_rot2x1_matrix` function processes the `MSG_ROT2x1_MATRIX` message. This message is sent by the Simulation Host to update the `RTS4x3` matrix in a configuration node when the only change is rotation around a single axis.

The function call is `msg_rot2x1_matrix(msgp)`, where `msgp` is a pointer to the `MSG_ROT2x1_MATRIX` message.

The function does the following:

- Determines the rotation axis from the message.
- If the rotation axis is 0, calls `vpt_update_2x1_heading`.
- If the rotation axis is 1, calls `vpt_update_2x1_pitch`.
- If the rotation axis is 2, calls `vpt_update_2x1_roll`.

Called By:	<code>process_a_msg</code>	
Routines Called:	<code>vpt_update_2x1_heading</code> <code>vpt_update_2x1_pitch</code> <code>vpt_update_2x1_roll</code>	
Parameters:	<code>MSG_ROT2x1_MATRIX</code>	<code>*msgp</code>
Returns:	<code>none</code>	

2.12.18.4 msg_rts4x3_matrix

The `msg_rts4x3_matrix` function processes the `MSG_RTS4x3_MATRIX` message. This message is sent by the Simulation Host to completely replace the transformation matrix in a configuration node defined with a matrix type of `RTS4x3`.

The function call is `msg_rts4x3_matrix(msgp)`, where `msgp` is a pointer to the `MSG_RTS4x3_MATRIX` message. The function calls `vpt_update_4x3_matrix` to process the change.

Called By:	process_a_msg	
Routines Called:	vpt_update_4x3_matrix	
Parameters:	MSG_RTS4x3_MATRIX	*msgp
Returns:	none	

2.12.18.5 msg_hprxyzs_matrix

The `msg_hprxyzs_matrix` function processes the `MSG_HPRXYZS_MATRIX` message. This message is sent by the Simulation Host to completely replace the transformation matrix in a configuration node defined with a matrix type of `HPRXYZS`.

The function call is `msg_hprxyzs_matrix(msgp)`, where `msgp` is a pointer to the `MSG_HPRXYZS_MATRIX` message. The function calls `vpt_update_hprxyzs` to process the change.

Called By:	process_a_msg	
Routines Called:	vpt_update_hprxyzs	
Parameters:	MSG_RTS4x3_MATRIX	*msgp
Returns:	none	

2.12.18.6 msg_translation

The `msg_translation` function processes the `MSG_TRANSLATION` message. This message is sent by the Simulation Host to update the translation portion of the matrix of a configuration node defined with a matrix type of `HPRXYZS`.

The function call is **msg_translation(msgp)**, where *msgp* is a pointer to the MSG_TRANSLATION message. The function calls **vpt_update_translation** to process the change.

Called By: process_a_msg

Routines Called: vpt_update_translation

Parameters: MSG_TRANSLATION *msgp

Returns: none

2.12.18.7 msg_scale

The **msg_scale** function processes the MSG_SCALE message. This message is sent by the Simulation Host to update the scale portion of the matrix of a configuration node defined with a matrix type of HPRXYZS.

The function call is **msg_scale(msgp)**, where *msgp* is a pointer to the MSG_SCALE message. The function calls **vpt_update_scale** to process the change.

Note: The MSG_SCALE message is reserved for future expansion.

Called By: process_a_msg

Routines Called: vpt_update_scale

Parameters: MSG_SCALE *msgp

Returns: none

2.12.18.8 msg_1rotation

The **msg_1rotation** function processes the MSG_1ROTATION message. This message is sent by the Simulation Host to update the matrix of a configuration node defined with a matrix type of HPRXYZS, when the only change is rotation around a single axis.

The function call is **msg_1rotation(msgp)**, where *msgp* is a pointer to the MSG_1ROTATION message. The function does the following:

- Determines the rotation axis from the message.
- If the rotation axis is 0, calls **vpt_update_heading**.
- If the rotation axis is 1, calls **vpt_update_pitch**.
- If the rotation axis is 2, calls **vpt_update_roll**.

Called By: process_a_msg

Routines Called: vpt_update_heading
vpt_update_pitch
vpt_update_roll

Parameters: MSG_1ROTATION *msgp

Returns: none

2.12.18.9 msg_3rotations

The msg_3rotations function processes the MSG_3ROTATIONS message. This message is sent by the Simulation Host to update the heading, pitch, and roll in the matrix of a configuration node defined with a matrix type of HPRXYZS.

The function call is **msg_3rotations(msgp)**, where *msgp* is a pointer to the MSG_3ROTATIONS message. The function calls vpt_update_hpr to process the change.

Called By: process_a_msg

Routines Called: vpt_update_hpr

Parameters: MSG_3ROTATIONS *msgp

Returns: none

2.12.19 print_msg.c

The print_msg.c CSU contains functions that can be used to print the contents of all messages passed between the CIG and the Simulation Host. These functions are provided primarily for testing and debugging. At the current time, these functions are used only to display messages to stdout; this feature is called debug message display.

The print_msg.c CSU contains the following functions:

- print_msg_* (one function for each message type)
- init_print_msg_array

The debug message display feature is enabled by setting the dr1lw_init_out debug flag to TRUE. This flag, initialized to FALSE, can be toggled using an option on the Gossip main menu.

Each message type can be individually enabled for display. When gossip starts up, it enables all message types. Using an option on the Gossip main menu, the user can access a sub-menu which provides the ability to disable all or selected message types.

The `init_print_msg_array` function initializes an array of all message types, indexed by message code. Each element contains the following:

- A boolean flag indicating whether or not the message type is enabled for debug display/printing during all states other than simulation.
- A boolean flag indicating whether or not the message type is enabled for debug display/printing during a runtime simulation.
- The message type.
- The name of the `print_msg_*` function used to print that message type.

The `host_if_debug_init` function (in the Host Interface Manager CSC) calls `init_print_msg_array` to initialize the array when gossip starts up. `host_if_debug_init` uses `*message_enabled` to point to the array returned by `init_print_msg_array`.

If debug message display is enabled, each frame's messages are written to a buffer in memory by `cigsimio_write` (in the Real-Time Processing CSC). At the end of each frame, the `host_if_debug_tick` function examines each message in the `cigsimio` buffer, and indexes into `*message_enabled` to see if that message type is enabled for debug display. If it is, `host_if_debug_tick` calls the `print_msg_*` function specified in that element of the `print_msg_array`. The contents of the message are then printed to `stdout`.

The messages buffered by `cigsimio_write` can also be written to a disk file. This process, called message recording, is enabled through Gossip and handled by the `cigsimio` functions. Refer to the Real-Time Processing CSC for more information. The recorded messages can be played back to repeat a simulation exercise; see the Stand-Alone Host Emulator (Flea) CSC for details.

For more information on `host_if_debug_init` and `host_if_debug_tick`, see the Host Interface Manager CSC.

2.12.19.1 `print_msg_*`

Each `print_msg_*` function prints the contents of a specific message type to a specified file. For example, the `print_msg_2d_setup` function prints `MSG_2D_SETUP` messages. The `print_msg_array` set up by `init_print_msg_array` is used to determine which `print_msg_*` function to call for each message type encountered.

At the current time, the `print_msg_*` functions are used only by `host_if_debug_tick`, to output all or selected messages to `stdout` each frame. Because they are tailored to specified message types, the display can be formatted to identify each parameter in the message. For more information on debug message display, refer to the Host Interface Manager CSC.

The function call for each function is `print_msg_<message_type>(listing_fp, msg_P, msg_type, msg_len)`, where:

listing_fp is a pointer to the file to which the message contents are to be written; for
host_if_debug_tick, this is always set to stdout
msg_P is a pointer to the message
msg_type is the message type
msg_len is the message length

Most of the routines are called indirectly by host_if_debug_tick, which indexes into the print_msg_array using the *message_enabled pointer. A few are called directly.

All of the routines call fprintf to write the contents of the message to the file specified by *listing_fp*. The routines index into the print_msg_array to get the text for the message type, then output the other information from the message with appropriate headings. Most of the functions return the message length as *msg_len*.

The following table lists the print_msg_* functions and identifies the SIM-to-CIG or CIG-to-SIM message each is used to print. The table also shows the function(s) that call and are called by each routine.

print_msg Function	Message Type Printed	Calls	Called By
print_msg_1rotation	MSG_1ROTATION	fprintf	*message_enabled
print_msg_2d_setup	MSG_2D_SETUP	fprintf	*message_enabled
print_msg_3rotations	MSG_3ROTATIONS	fprintf	*message_enabled
print_msg_add_traj_table	MSG_ADD_TRAJ_TABLE	fprintf	*message_enabled
print_msg_agl	MSG_AGL	fprintf	*message_enabled
print_msg_agl_setup	MSG_AGL_SETUP	fprintf	*message_enabled
print_msg_ammo_define	MSG_AMMO_DEFINE	fprintf	*message_enabled
print_msg_calibration_image	MSG_CALIBRATION_IMAGE	fprintf	*message_enabled
print_msg_cancel_round	MSG_CANCEL_ROUND	fprintf	*message_enabled
print_msg_cig_ctl	MSG_CIG_CTL	fprintf	*message_enabled
print_msg_cloud_state	MSG_CLOUD_STATE	fprintf	*message_enabled
print_msg_create_confignode	MSG_CREATE_CONFIGNODE	fprintf	*message_enabled
print_msg_default	(any unknown message type)	fprintf	*message_enabled
print_msg_define_tx_mode	MSG_DEFINE_TX_MODE	fprintf	*message_enabled
print_msg_delete_traj_table	MSG_DELETE_TRAJ_TABLE	fprintf	*message_enabled
print_msg_dr11_pkt_size	MSG_DR11_PKT_SIZE	fprintf	*message_enabled
print_msg_end	MSG_END	fprintf	*message_enabled
print_msg_file_descr	MSG_FILE_DESCR	fprintf	*message_enabled
print_msg_file_status	MSG_FILE_STATUS	fprintf	*message_enabled
print_msg_file_xfer	MSG_FILE_XFER	fprintf	*message_enabled
print_msg_gun_overlay	MSG_GUN_OVERLAY	fprintf	*message_enabled
print_msg_hit_return	MSG_HIT_RETURN	fprintf	*message_enabled
print_msg_hit_return48	MSG_HIT_RETURN48	fprintf	*message_enabled
print_msg_hprxyzs_matrix	MSG_HPRXYZS_MATRIX	fprintf	*message_enabled
print_msg_laser_return	MSG_LASER_RETURN	fprintf	*message_enabled
print_msg_local_terrain	MSG_LOCAL_TERRAIN	fprintf	print_msg_lt_ piece
print_msg_lt_piece	MSG_LT_PIECE	fprintf, print_msg_ local_terrain	host_if_debug_ tick
print_msg_lt_state	MSG_LT_STATE	fprintf	*message_enabled
print_msg_miss	MSG_MISS	fprintf	*message_enabled
print_msg_obscure	MSG_OBSCURE	fprintf	*message_enabled
print_msg_otherveh_state	MSG_OTHERVEH_STATE	fprintf	*message_enabled
print_msg_overlay_setup	MSG_OVERLAY_SETUP	fprintf	*message_enabled
print_msg_pass_back	MSG_PASS_BACK	fprintf	*message_enabled
print_msg_pass_on	MSG_PASS_ON	fprintf	*message_enabled
print_msg_ppm_display_mode	MSG_PPM_DISPLAY_MODE	fprintf	*message_enabled
print_msg_ppm_display_offset	MSG_PPM_DISPLAY_OFFSET	fprintf	*message_enabled

print_msg_ppm_pixel_location	MSG_PPM_PIXEL_LOCATION	fprintf	*message_enabled
print_msg_ppm_pixel_state	MSG_PPM_PIXEL_STATE	fprintf	*message_enabled
print_msg_process_chord	MSG_PROCESS_CHORD	fprintf	*message_enabled
print_msg_process_round	MSG_PROCESS_ROUND	fprintf	*message_enabled
print_msg_process_round48	MSG_PROCESS_ROUND48	fprintf	*message_enabled
print_msg_request_laser_range	MSG_REQUEST_LASER_RANGE	fprintf	*message_enabled
print_msg_request_point_info	MSG_REQUEST_POINT_INFO	fprintf	*message_enabled
print_msg_return_point_info	MSG_RETURN_POINT_INFO	fprintf	*message_enabled
print_msg_rot2x1_matrix	MSG_ROT2x1_MATRIX	fprintf	*message_enabled
print_msg_round_fired	MSG_ROUND_FIRED	fprintf	*message_enabled
print_msg_rts4x3_matrix	MSG_RTS4x3_MATRIX	fprintf	*message_enabled
print_msg_scale	MSG_SCALE	fprintf	*message_enabled
print_msg_shot_report	MSG_SHOT_REPORT	fprintf, printf	*message_enabled
print_msg_shot_report48	MSG_SHOT_REPORT48	fprintf	*message_enabled
print_msg_show_effect	MSG_SHOW_EFFECT	fprintf	*message_enabled
print_msg_sio_close	MSG_SIO_CLOSE	fprintf	*message_enabled
print_msg_sio_init	MSG_SIO_INIT	fprintf	*message_enabled
print_msg_sio_write	MSG_SIO_WRITE	fprintf, putchar	*message_enabled
print_msg_staticveh_rem	MSG_STATICVEH_REM	fprintf	*message_enabled
print_msg_staticveh_state	MSG_STATICVEH_STATE	fprintf	*message_enabled
print_msg_subsys_mode	MSG_SUBSYS_MODE	fprintf	*message_enabled
print_msg_sys_error	MSG_SYS_ERROR	fprintf	*message_enabled
print_msg_test_name	MSG_TEST_NAME	fprintf	*message_enabled
print_msg_tf_hdr	MSG_TF_HDR	fprintf	*message_enabled
print_msg_tf_init_hdr	MSG_TF_INIT_HDR	fprintf	*message_enabled
print_msg_tf_init_pt	MSG_TF_INIT_PT	fprintf	*message_enabled
print_msg_tf_pt	MSG_TF_PT	fprintf	*message_enabled
print_msg_tf_pt48	MSG_TF_PT48	fprintf	*message_enabled
print_msg_tf_state	MSG_TF_STATE	fprintf	*message_enabled
print_msg_tf_vehicle_pos	MSG_TF_VEHICLE_POS	fprintf	*message_enabled
print_msg_traj_chord	MSG_TRAJ_CHORD	fprintf	*message_enabled
print_msg_traj_entry	MSG_TRAJ_ENTRY	fprintf	*message_enabled
print_msg_traj_entry_xfer	MSG_TRAJ_ENTRY_XFER	fprintf	*message_enabled
print_msg_traj_table_xfer	MSG_TRAJ_TABLE_XFER	fprintf	*message_enabled
print_msg_translation	MSG_TRANSLATION	fprintf	*message_enabled
print_msg_view_flags	MSG_VIEW_FLAGS	fprintf	*message_enabled
print_msg_view_magnification	MSG_VIEW_MAGNIFICATION	fprintf	*message_enabled
print_msg_viewport_state	MSG_VIEWPORT_STATE	fprintf	*message_enabled
print_msg_viewport_update	MSG_VIEWPORT_UPDATE	fprintf	*message_enabled

Called By: see table above
(most functions are called by `host_if_debug_tick` via
`*message_enabled[msg_type].print_msg`)

Routines Called: see table above
(most functions call only `fprintf`)

Parameters: FILE `*listing_fp`
`<message_type>` `*msg_P`
 INT_2 `msg_type`
 INT_2 `msg_len`

Returns: `print_msg_local_terrain` returns
`(msg_len + poly_byte_count + bvol_byte_count)`
 all other functions return `msg_len`

2.12.19.2 `init_print_msg_array`

The `init_print_msg_array` function initializes and returns the `print_msg_array` that contains the enabled flags and specifies the `print_msg_*` function for each valid message type. The `host_if_debug_tick` function indexes into the returned array to call the appropriate `print_msg_*` function for each message to be printed. The `print_msg_*` functions index into this array to get the message name.

The function call is `init_print_msg_array(last_msg)`, where `last_msg` is a pointer to the highest (largest) defined message code.

`init_print_msg_array` initializes the array elements as follows:

- The enabled flag for all states other than runtime is set to TRUE.
- The enabled flag for runtime is set to FALSE.
- The message name is set to "UNUSED_."
- The function pointer is set to `print_msg_default`.

It then sets the name and function pointer appropriately for each defined message code.

Called By: `host_if_debug_init`

Routines Called: none

Parameters: INT_2 `*last_msg`

Returns: `print_msg_array`